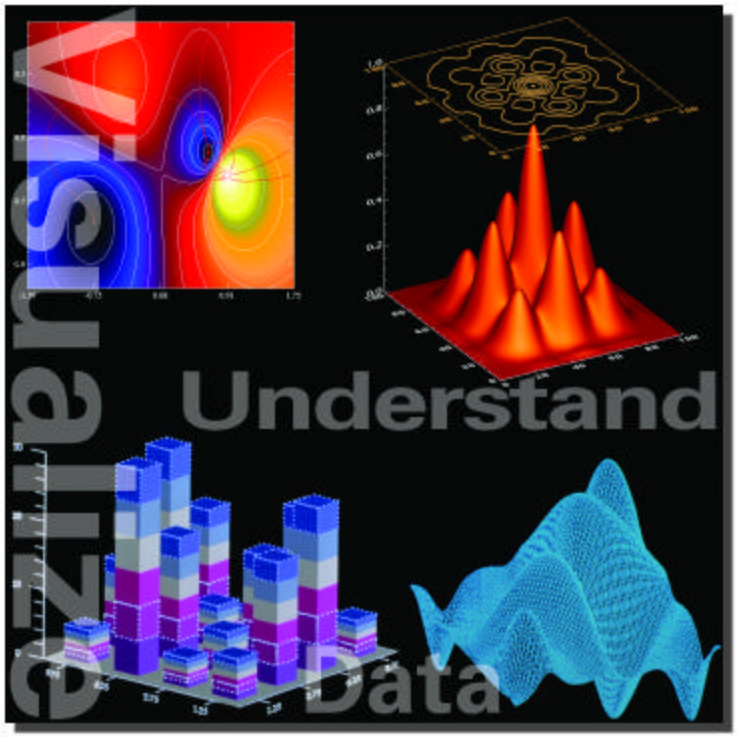
PV-WAVE 7.5®



IMSL Mathematics Reference

HELPING CUSTOMERS SOLVE COMPLEX PROBLEMS

Visual Numerics, Inc.

Visual Numerics, Inc.
2500 Wilcrest Drive
Suite 200
Houston, Texas 77042-2579
United States of America
713-784-3131
800-222-4675
(FAX) 713-781-9260
http://www.vni.com
e-mail: info@boulder.vni.com

Visual Numerics, Inc. (France) S.A.R.L.
Tour Europe
33 place des Corolles
Cedex 07
92049 PARIS LA DEFENSE
FRANCE
+33-1-46-93-94-20
(FAX) +33-1-46-93-94-39
e-mail: info@vni-paris.fr

Visual Numerics International, Ltd.
Suite 1
Centennial Court
East Hampstead Road
Bracknell, Berkshire
RG 12 1 YQ
UNITED KINGDOM
+01-344-458-700
(FAX) +01-344-458-748
e-mail: info@vniuk.co.uk

Visual Numerics, Inc.
7/F, #510, Sect. 5
Chung Hsiao E. Rd.
Taipei, Taiwan 110 ROC
+886-2-727-2255
(FAX) +886-2-727-6798
e-mail: info@vni.com.tw

Visual Numerics International GmbH
Zettachring 10
D-70567 Stuttgart
GERMANY
+49-711-13287-0
(FAX) +49-711-13287-99
e-mail: info@visual-numerics.de

Visual Numerics Japan, Inc.
Gobancho Hikari Building, 4th Floor
14 Gobancho
Chiyoda-Ku, Tokyo, 102
JAPAN
+81-3-5211-7760
(FAX) +81-3-5211-7769
e-mail: vda-sprt@vnij.co.jp

VIsual Numerics S.A. de C.V.
Cerrada de Berna 3, Tercer Piso
Col. Juarez
Mexico, D.F. C.P. 06600
Mexico

Visual Numerics, Inc., Korea
Rm. 801, Hanshin Bldg.
136-1, Mapo-dong, Mapo-gu
Seoul 121-050
Korea

Table of Contents

Preface

PV-WAVE:IMSL Mathematics is a powerful tool for mathematical, statistical, and scientific computing. This *PV-WAVE:IMSL Mathematics Reference* documents the routines that support this functionality. Each function and procedure is designed for use in research as well as in technical applications.

Finding the Appropriate Routine

This *PV-WAVE:IMSL Mathematics Reference* is organized into 12 chapters. Each chapter groups routines with similar computational or analytical capabilities. To locate the appropriate function for a given problem, refer to the *Contents of Chapter* subsection in the introduction to each chapter or the alphabetical *Summary of Routines* in Appendix B.

Often the quickest way to use this *PV-WAVE:IMSL Mathematics Reference* is to find an example similar to your problem and mimic the example. Documented routines contain at least one example.

Documentation Organization

Each PV-WAVE:IMSL Mathematics routine conforms to established conventions in programming and documentation. The uniform design of these routines makes it easy to use more than one function or procedure in a given application. Also, the design consistency enables you to apply your experience with one

PV-WAVE:IMSL Mathematics function to all other PV-WAVE:IMSL Mathematics functions.

This manual contains a concise description of each function and procedure. Each chapter begins with an introduction containing a *Contents of Chapter* that lists the routines discussed in that chapter and the corresponding page numbers. At least one example, including sample input and results, is provided for most routines. The documentation for each routine contains of the following information:

- **Routine Name** — procedure or function name with purpose statement
- **Usage** — calling sequence
- **Input/Output Parameters** — description of the parameters in the order of their occurrence

  Input — parameter must be initialized; it is not changed by the function

  Input/Output — parameter must be initialized; the routine returns output through the parameter; the parameter cannot be a constant or an expression

  Output — no initialization is necessary; the routine returns output through this parameter; the parameter cannot be a constant or an expression
- **Returned Value** — value returned by the function
- **Keywords** — description of keywords available for a particular routine
- **Discussion** — discussion of the algorithm and references to detailed information
- **Examples** — one or more examples showing applications of this routine using the required parameters
- **Errors** — list of errors that may occur with a particular routine for which a user-defined action may be desired

References

References are listed alphabetically by author in Appendix A, *References*.

Typographical Conventions

The following typographical conventions are used in this guide:

- PV-WAVE:IMSL Mathematics code examples appear in a typewriter font. For example:

```
PLOT, temp, s02, Title = 'Air Quality'
```

- Comments for commands and program examples are shown in the following manner:

```
PLOT, temp, s02, Title = 'Air Quality'
   ; This is a comment for the PLOT command.
```

  Comments are used often to explain code fragments and examples.

- PV-WAVE:IMSL Mathematics commands are not case-sensitive. However, in this manual, variables are shown in lowercase italics (*myvar*), function and procedure names are shown in uppercase (XYOUTS), keywords are shown in mixed case italic (*XTitle*), and system variables are shown in regular mixed case type (!Version).

- A $ at the end of a PV-WAVE:IMSL Mathematics line indicates that the current statement is continued on the following line.

  This means, for instance, that *strings* are never split onto two lines without the addition of the string concatenation operator (+) or a comma in some cases. For example, the following lines would produce an error if entered literally in PV-WAVE:IMSL Mathematics:

```
WAVE> PLOT, x, y, Title = 'Average $
   Air Temperatures by Two-Hour Periods'
      ; Note that the string is split onto two lines. This syntax would
      ; produce an error.
```

  The correct way to enter these lines is:

```
WAVE> PLOT, x, y, Title = 'Average ' + $
   'Air Temperatures by Two-Hour Periods'
      ; This is the correct way to split a string onto two command
      ; lines.
```

  The string concatenation symbol (+) is used at the end of the first line, and the split portions of the string are enclosed by delimiters. This is the convention used in this reference whenever a string spans two lines. This is still only one command, even though multiple lines are used.

- Reserved words, such as FOR, IF, and CASE, are always shown in capital letters.

Technical Support

If you have problems installing, unlocking, or running your software, contact Visual Numerics Technical Support by calling:

Office Location	Phone Number
Corporate Headquarters Houston, Texas	713-784-3131
Boulder, Colorado	303-939-8920
France	+33-1-46-93-94-20
Germany	+49-711-13287-0
Japan	+81-3-5211-7760
Korea	+82-2-3273-2633
Mexico	+52-5-514-9730
Taiwan	+886-2-727-2255
United Kingdom	+44-1-344-458-700

Users outside the U.S., France, Germany, Japan, Korea, Mexico, Taiwan, and the U.K. can contact their local agents.

Please be prepared to provide the following information when you call for consultation during Visual Numerics business hours:

- Your license number, a six-digit number that can be found on the packing slip accompanying this order. (If you are evaluating the software, just mention that you are from an evaluation site.)
- The name and version number of the product. For example, PV-WAVE 7.0.
- The type of system on which the software is being run. For example, SPARCstation, IBM RS/6000, HP 9000 Series 700.
- The operating system and version number. For example, HP-UX 10.2 or IRIX 6.5.
- A detailed description of the problem.

FAX and E-mail Inquiries

Contact Visual Numerics Technical Support staff by sending a FAX to:

Office Location	FAX Number
Corporate Headquarters	713-781-9260
Boulder, Colorado	303-245-5301
France	+33-1-46-93-94-39
Germany	+49-711-13287-99
Japan	+81-3-5211-7769
Korea	+82-2-3273-2634
Mexico	+52-5-514-4873
Taiwan	+886-2-727-6798
United Kingdom	+44-1-344-458-748

or by sending E-mail to:

Office Location	E-mail Address
Boulder, Colorado	`support@boulder.vni.com`
France	`support@vni-paris.fr`
Germany	`support@visual-numerics.de`
Japan	`vda-sprt@vnij.co.jp`
Korea	`support@vni.co.kr`
Taiwan	`support@vni.com.tw`
United Kingdom	`support@vniuk.co.uk`

Electronic Services

Service	Address
General e-mail	`info@boulder.vni.com`
Support e-mail	`support@boulder.vni.com`
World Wide Web	`http://www.vni.com`
Anonymous FTP	`ftp.boulder.vni.com`
FTP Using URL	`ftp://ftp.boulder.vni.com/VNI/`
PV-WAVE Mailing List:	`Majordomo@boulder.vni.com`
To subscribe include:	`subscribe pv-wave YourEmailAddress`
To post messages	`pv-wave@boulder.vni.com`

Introduction

Starting PV-WAVE:IMSL Mathematics

To start PV-WAVE:IMSL Mathematics, you must first be running PV-WAVE. For detailed information on starting PV-WAVE, see the *PV-WAVE User's Guide* or the installation instructions.

At the `WAVE>` prompt, type:

```
@math_startup
```

You will then see this message:

```
PV-WAVE:IMSL Mathematics Toolkit is initialized.
```

You are now ready to use PV-WAVE:IMSL Mathematics.

Rows vs Columns

In this book we use the following convention for 2D arrays: "row" refers to the first index of the array and "column" refers to the second. So for a 2D array *A*, *A*(*i*,*j*) is the element in row *i* and column *j*. The PM command makes this easy to visualize:

```
a = INTARR( 4, 8 )    &    a(2,5) = 1    &    PM, a
   0       0       0       0       0       0       0       0
   0       0       0       0       0       0       0       0
   0       0       0       0       0       1       0       0
   0       0       0       0       0       0       0       0
```

Underflow and Overflow

In most cases, PV-WAVE:IMSL Mathematics routines are written so that computations are not affected by underflow, provided the system (hardware or software) replaces an underflow with the value zero. Normally, system error messages indicating underflow can be ignored.

PV-WAVE:IMSL Mathematics routines also are written to avoid overflow. A program that produces system error messages indicating overflow should be examined for programming errors such as incorrect input data, mismatch of parameter types, or improper dimensions.

In many cases, the documentation for a function points out common pitfalls that can lead to failure of the algorithm.

Missing Values

Some of the routines in this *PV-WAVE:IMSL Mathematics Reference* allow the data to contain missing values. These routines recognize as a missing value the special value referred to as "Not a Number" or NaN. The actual value varies on different computers, but it can be obtained by reference to function MACHINE.

The manner in which missing values are treated depends on the individual function as described in the documentation for that function.

User Errors

PV-WAVE:IMSL Mathematics mathematical functions attempt to detect user errors and handle them in a way that provides as much information to the user as possible. To do this, five levels of *Informational Error* severity, in addition to the basic PV-WAVE:IMSL Mathematics error-handling facility, are recognized. Following a call to a PV-WAVE:IMSL Mathematics mathematical or statistical function, the system variables !Error and !Cmast_Err contain information concerning the current error state. The system variable !Error contains the error *number* of the last error. System variable !Cmast_Err is set to either zero, which indicates that an *Informational Error* did not occur, or to the error code of the last *Informational Error* that did occur.

Errors and Default Actions

When your application returns from a PV-WAVE:IMSL Mathematics mathematical function, the system variable !Cmast_Err is set either to zero, which indicates

that an informational error did not occur, or to the error code for the last *Informational Error* that did occur. Internally, there are five levels of *Informational Error* severity: note, alert, warning, fatal, and terminal. Although PV-WAVE:IMSL Mathematics does not allow users to directly manipulate how these errors are interpreted internally, some control over the output of error messages is allowed. All informational error messages are printed by default. Setting the system variable !Quiet to a nonzero value suppresses output of notes, alerts, and warnings.

The system variable !Error remains active during all PV-WAVE:IMSL Mathematics error states. But when an *Informational Error* occurs within a mathematical function, the system variable !Cmast_Err is used.

What Determines Error Severity

Although your input(s) may be mathematically correct, limitations of the computer's arithmetic and the algorithm itself can make it impossible to accurately compute an answer. In this case, the assessed degree of accuracy determines the severity of the error. In instances where the function computes several output quantities and some are not computable, an error condition exists. Its severity depends on an assessment of the overall impact of the error.

Functions for Error Handling

With respect to *Informational Errors*, you can interact with the PV-WAVE:IMSL Mathematics error-handling system in two ways: (1) change the default printing actions and (2) determine the code of an *Informational Error* in order to take corrective action. To change the default printing action, set the system variable !Quiet to a nonzero value. Use CMAST_ERR_TRANS to retrieve the integer code for an informational error.

Use of CMAST_ERR_TRANS to Determine Program Action

In the program segment below, the Cholesky factorization of a matrix is to be performed. If it is determined that the matrix is not nonnegative definite (and often this is not immediately obvious), the program takes a different branch:

```
x = CHNNDFAC, a, fac
    ; Call CHNNDFAC with a matrix that may not be nonnegative definite.


IF (CMAST_ERROR_TRANS($
    'MATH_NOT_NONNNEG_DEFINITE') eq !Cmast_Err)) THEN ...
        ; Check the system variable !Cmast_Err to see if it contains the
        ; error code for the error NOT_NONNNEG_DEFINITE.
```

CHAPTER 1

Linear Systems

Contents of Chapter

Sparse Matrices

Matrix Inversion

Function INV is used to invert an $n \times n$ nonsingular matrix—either real or complex. The inverse also can be obtained by using keyword *Inverse* in functions for solving systems of linear equations. The inverse of a matrix need not be computed if the purpose is to solve one or more systems of linear equations. Even with multiple right-hand sides, solving a system of linear equations by computing the inverse and performing matrix multiplication is usually more expensive than the method discussed in the next section.

Solving Systems of Linear Equations

A square system of linear equations has the form $Ax = b$, where A is a user-specified $n \times n$ matrix, b is a given n-vector, and x is the solution n-vector. Each entry of A and b must be specified by the user. The entire vector x is returned as output.

When A is invertible, a unique solution to $Ax = b$ exists. The most commonly used direct method for solving $Ax = b$ factors the matrix A into a product of triangular matrices and solves the resulting triangular systems of linear equations. Functions that use direct methods for solving systems of linear equations all compute the solution to $Ax = b$. PV-WAVE:IMSL Mathematics functions LUSOL, CHSOL, and CHNNDSOL can be used to compute x.

Matrix Factorizations

In some applications, you may only want to factor the $n \times n$ matrix A into a product of two triangular matrices. Functions and procedures that end with "FAC" are designed to compute these factorizations. Suppose that in addition to the solution x of a linear system of equations $Ax = b$, you want the LU factorization of A. First, use the LUFAC procedure to obtain the LU factorization in a condensed format; then, call LUSOL with this factorization and a right-hand side b to compute the solution. If the factorization is desired in separate, full matrices, the LUFAC procedure can be called with keywords L and U to return L and U separately.

Besides the basic matrix factorizations, such as LU and LL^T, additional matrix factorizations also are provided. For a real matrix A, QR factorization can be

computed by the QRFAC procedure. Functions for computing the Singular Value Decomposition (SVD) of a matrix are discussed in a later section.

Multiple Right-hand Sides

Consider the case in which a system of linear equations has more than one right-hand side vector. It is most economical to find the solution vectors by first factoring the coefficient matrix A into products of triangular matrices. Then, the resulting triangular systems of linear equations are solved for each right-hand side. When A is a real general matrix, access to the LU factorization of A is computed by using the LUFAC procedure. The solution x_k for the k-th right-hand side vector b_k is then found by two triangular solves, $Ly_k = b_k$ and $Ux_k = y_k$. The LUSOL function is called with the computed factorization and is used to solve each right-hand side. This process can be followed when using other functions for solving systems of linear equations.

Least-squares Solution and QR Factorization

Least-squares solutions are usually computed for an over-determined system of linear equations $A_{m \times n}\ x = b$, where $m > n$. A least-squares solution x minimizes the Euclidean length of the residual vector $r = Ax - b$. The QRSOL function computes a unique least-squares solution for x when A has full-column rank. If A is rank-deficient, then the *base* solution for some variables is computed. These variables consist of the resulting columns after the interchanges. The QR decomposition, with column interchanges or pivoting, is computed such that $AP = QR$. Here, Q is orthogonal, R is upper-trapezoidal with its diagonal elements nonincreasing in magnitude, and P is the permutation matrix determined by the pivoting. The base solution x_B is obtained by solving $R(P^T)x = Q^Tb$ for the base variables. For details, see the *Discussion* of QRSOL on page 30. The QR factorization of a matrix A, such that $AP = QR$ with P specified by the user, can be computed using the QRFAC procedure.

Singular Value Decomposition and Generalized Inverse

The SVD of an m by n matrix A is a matrix decomposition, $A = USV^T$. With $q = \min(m, n)$, the factors $U_{m \times q}$ and $V_{n \times q}$ are orthogonal matrices, and $S_{q \times q}$ is a nonnegative diagonal matrix with nonincreasing diagonal terms. The SVDCOMP function computes the singular values of A by default. By using keywords, part or all of the U and V matrices, an estimate of the rank of A, and the generalized inverse of A also can be obtained.

Ill-conditioning and Singularity

An $m \times n$ matrix A is mathematically singular if there is an $x \neq 0$ such that $Ax = 0$. In this case, the system of linear equations $Ax = b$ does not have a unique solution. On the other hand, a matrix A is *numerically* singular if it is "close" to a mathematically singular matrix. Such problems are called *ill-conditioned*. If the numerical results with an ill-conditioned problem are unacceptable, either use more accuracy if it is available (for type *float* accuracy switch to *double*) or obtain an approximate solution to the system. One form of approximation can be obtained using the SVD of A: If $q = \min(m, n)$ and

$$A = \sum_{i=1}^{q} s_{i,i} u_i v_i^T$$

then the approximate solution is given by the following:

$$x_k = \sum_{i=1}^{k} t_{i,i} (b^T u_i) v_i$$

The scalars $t_{i,i}$ are defined by

$$t_{i,i} = \begin{cases} s_{i,i}^{-1} & \text{if } s_{i,i} \geq tol \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The user specifies the value of *tol*. This value determines how "close" the given matrix is to a singular matrix. Further restrictions may apply to the number of terms in the sum, $k \leq q$. For example, there may be a value of $k \leq q$ such that the scalars $|(b^T u_i)|$, $i > k$, are smaller than the average uncertainty in the right-hand side b. This means that these scalars can be replaced by zero, and hence, b is replaced by a vector that is within the stated uncertainty of the problem.

Notation

Since many functions and procedures described in this chapter operate on both real or complex matrices, the notation A^H is used to represent both the transpose of A if A is real and the conjugate transpose if A is complex.

Sparse Matrices: Direct Methods

A number of routines are provided that employ direct methods (as opposed to iterative methods) for solving problems involving sparse matrices.

For general sparse linear systems, SP_LUFAC and SP_LUSOL form a factor/solve function pair. If a sparse matrix the problem $Ax = b$ is to be solved for a single A, but multiple right-hand sides, b, then SP_LUFAC should first be used to compute an LU decomposition of A, then follow multiple calls to SP_LUSOL (one for each right-hand side, b). If only one right-hand side, b, is involved then SP_LUSOL can be called directly, in which case the factor step is computed internally by SP_LUSOL.

For general banded systems, SP_BDSOL and SP_BDFAC form a factor/solve pair. The relationship between SP_BSFAC and SP_BDSOL is similar to that of SP_LUFAC and SP_LUSOL.

The functions SP_PDFAC and SP_PDSOL are provided for working with systems involving sparse symmetric positive definite matrices.The relationship between SP_PDFAC and SP_PDSOL is similar to that of SP_LUFAC and SP_LUSOL.

The functions SP_BDDFAC and SP_BDPDSOL are provided for working with systems involving banded symmetric positive definite matrices.The relationship between SP_BDPDFAC and SP_BDPDSOL is similar to that of SP_LUFAC and SP_LUSOL.

Direct Methods

SP_LUFAC	*LU* factorization of general matrices
SP_LUSOL	Systems involving general matrices
SP_BDFAC	*LU* factorization of band matrices
SP_BDSOL	Systems involving band matrices
SP_PDFAC	Factorization of symmetric positive definite matrices
SP_PDSOL	Systems involving symmetric positive definite matrices
SP_BDPDFAC	Cholesky factorization of symmetric positive definite matrices in band symmetric storage mode
SP_BDPDSOL	Systems involving symmetric positive definite matrices in band symmetric storage mode

Sparse Matrices: Iterative Methods

Two routines are provided that employ iterative methods (as opposed to direct methods) for solving problems involving sparse matrices.

The function SP_GMRES, based on the FORTRAN subroutine GMRESD by H. F. Walker, solves the linear system $Ax = b$ using the GMRES method. This method is described in detail by Saad and Schultz (1986) and Walker (1988).

The function SP_CG solves the symmetric definite linear system $Ax = b$ using the conjugate gradient method with optional preconditioning. This method is described in detail by Golub and Van Loan (1983, chapter 10), and in Hageman and Young (1981, chapter 7).

Iterative Methods

SP_GMRES	Restarted generalized minimum residual (GMRES) method
SP_CG	Conjugate gradient method

Sparse Matrices: Utilities

Utilities designed to aid with the manipulation of sparse matrices are also provided. The common operation of matrix-vector multiplication can be efficiently executed using the function SP_MVMUL.

Sparse Matrices: Matrix Storage Modes

The dense linear algebra functions in PV-WAVE require input consisting of matrix dimensions and all values for the matrix entries. Clearly, this is not practical for sparse linear algebra. Three different storage formats can be used for the functions in the sparse matrix sections. These methods include:

- Sparse coordinate storage format
- Band storage format
- Compressed sparse column (CSC) format

Sparse Coordinate Storage Format

Only the non-zero elements of a sparse matrix need to be communicated to a function. Sparse coordinate storage format stores the value of each matrix entry along with that entry's row and column index. The following system variables are defined to support this concept:

```
!F_SP_ELEM = {f_sp_elem_str, row:0L, col:0L,
                             val:float(0.0)}
!D_SP_ELEM = {d_sp_elem_str, row:0L, col:0L,
```

```
                                    val:double(0.0)}
!C_SP_ELEM = {c_sp_elem_str, row:0L, col:0L,
                                    val:complex(0.0)}
!Z_SP_ELEM = {z_sp_elem_str, row:0L, col:0L,
                                    val:dcomplex(0.0)}
```

As an example consider the 6 x 6 matrix:

$$A = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & -3 & -1 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 \\ -2 & 0 & 0 & -7 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{bmatrix}$$

The matrix *A* has 15 non-zero elements, and its sparse coordinate representation would be

```
row   0   1   1   1   2   3   3   3   4   4   4   4   5   5   5
col   0   1   2   3   2   0   3   4   0   3   4   5   0   1   5
val   2   9  -3  -1   5  -2  -7  -1  -1  -5   1  -3  -1  -2   6
```

Since this representation does not rely on order, an equivalent form would be

```
row   5   4   3   0   5   1   2   1   4   3   1   4   3   5   4
col   0   0   0   0   1   1   2   2   3   3   3   4   4   5   5
val  -1  -1  -2   2  -2   9   5  -3  -5  -7  -1   1  -1   6  -3
```

There are different ways this data could be used to initialize. Consider the following program fragment:

```
A = replicate(!F_sp_elem, 15)
a(*).row = [0, 1, 1, 1, 2, $
            3, 3, 3, 4, 4, $
            4, 4, 5, 5, 5]
```

```
a(*).col = [0, 1, 2, 3, 2, $
            0, 3, 4, 0, 3, $
           4, 5, 0, 1, 5]
a(*).val = [2, 9, -3, -1, 5,$
            -2, -7, -1, -1, -5, 1, $
            -3, -1, -2, 6]
B = replicate(!F_sp_elem, 15)
b(*).row = [5, 4, 3, 0, 5, $
            1, 2, 1, 4, 3, $
            1, 4, 3, 5, 4]
b(*).col = [0, 0, 0, 0, 1, $
            1, 2, 2, 3, 3, $
            3, 4, 4, 5, 5]
b(*).val = [-1, -1, -2, 2, -2,$
            9, 5, -3, -5, -7, -1, $
            1, -1, 6, -3]
```

Both `a` and `b` represent the sparse matrix *A*.

A sparse symmetric or Hermitian matrix is a special case, since it is only necessary to store the diagonal and either the upper or lower triangle. As an example, consider the 5 x 5 linear system:

$$H = \begin{bmatrix} 4 & 1-i & 0 & 0 \\ 1+i & 4 & 1-i & 0 \\ 0 & 1+i & 4 & 1-i \\ 0 & 0 & 1+i & 4 \end{bmatrix}$$

The Hermitian and symmetric positive definite system solvers in this module expect the diagonal and lower triangle to be specified. The sparse coordinate form for the lower triangle is given by

row	0	1	2	3	1	2	3
col	0	1	2	3	0	1	2
val	(4,0)	(4,0)	(4,0)	(4,0)	(1,1)	(1,1)	(1,1)

The following program fragment will initialize *H*.

```
A = replicate(!C_sp_elem, 7)
a(*).row = [0, 1, 2, 3, 1, 2, 3]
a(*).col = [0, 1, 2, 3, 0, 1, 2]
a(*).val = [COMPLEX(4, 0), COMPLEX(4, 0), $
            COMPLEX(4, 0), COMPLEX(4, 0), $
            COMPLEX(1, 1), COMPLEX(1, 1), $
            COMPLEX(1, 1)]
```

There are some important points to note here. Note that *H* is not symmetric, but rather Hermitian. The functions that accept Hermitian data understand this and operate assuming that

$$h_{ij} = \bar{h}_{ji}.$$

The Sparse Matrix Module cannot take advantage of the symmetry in matrices that are not positive definite. The implication here is that a symmetric matrix that happens to be indefinite cannot be stored in this compact symmetric form. Rather, both upper and lower triangles must be specified and the sparse general solver called.

Band Storage Format

A band matrix is an M x N matrix A with all of its non-zero elements "close" to the main diagonal. Specifically, values $A_{ij} = 0$ if $i - j > nlca$ or $j - i > nuca$. The integer m = `nlca` + `nuca` + 1 is the total band width. The diagonals, other than the main diagonal, are called codiagonals. While any `M x N` matrix is a band matrix, band storage format is only useful when the number of non-zero codiagonals is much less than *N*.

In band storage format, the `nlca` lower codiagonals and the `nuca` upper codiagonals are stored in the rows of an array of size m x N. The elements are stored in the same column of the array as they are in the matrix. The values A_{ij} inside the band width are stored in the linear array in positions `[(i-j+nuca+1)* n+j]`. This results in a row-major, one-dimensional mapping from the two-dimensional notion of the matrix.

For example, consider the 5 x 5 matrix *A* with 1 lower and 2 upper codiagonals:

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & 0 & 0 \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} & 0 \\ 0 & A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ 0 & 0 & A_{3,2} & A_{3,3} & A_{3,4} \\ 0 & 0 & 0 & A_{4,3} & A_{4,4} \end{bmatrix}$$

In band storage format, the data would be arranged as:

$$\begin{bmatrix} 0 & 0 & A_{0,2} & A_{1,3} & A_{2,4} \\ 0 & A_{0,1} & A_{1,2} & A_{2,3} & A_{3,4} \\ A_{0,0} & A_{1,1} & A_{2,2} & A_{3,3} & A_{4,4} \\ A_{1,0} & A_{2,1} & A_{3,2} & A_{4,3} & 0 \end{bmatrix}$$

This data would be then be stored contiguously, row-major order, in an array of length 20.

As an example, consider the following tridiagonal matrix:

$$A = \begin{bmatrix} 10 & 1 & 0 & 0 & 0 \\ 5 & 20 & 2 & 0 & 0 \\ 0 & 6 & 30 & 3 & 0 \\ 0 & 0 & 7 & 40 & 4 \\ 0 & 0 & 0 & 8 & 50 \end{bmatrix}$$

The following code segment will store this matrix in band storage format:

```
a = [0, 1, 2, 3, 4, $
     10, 20, 30, 40, 50, $
     5, 6, 7, 8, 0]
```

As in the sparse coordinate representation, there is a space saving symmetric version of band storage. As an example, we look at the following 5 x 5 symmetric problem:

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & 0 & 0 \\ A_{0,1} & A_{1,1} & A_{1,2} & A_{1,3} & 0 \\ A_{0,2} & A_{1,2} & A_{2,2} & A_{2,3} & A_{2,4} \\ 0 & A_{1,3} & A_{2,3} & A_{3,3} & A_{3,4} \\ 0 & 0 & A_{2,4} & A_{3,4} & A_{4,4} \end{bmatrix}$$

In band symmetric storage format, the data would be arranged as:

$$\begin{bmatrix} 0 & 0 & A_{0,2} & A_{1,3} & A_{2,4} \\ 0 & A_{0,1} & A_{1,2} & A_{2,3} & A_{3,4} \\ A_{0,0} & A_{1,1} & A_{2,2} & A_{3,3} & A_{4,4} \end{bmatrix}$$

The following Hermitian example illustrate the procedure:

$$H = \begin{bmatrix} 8 & 1+i & 1+i & 0 & 0 \\ 1-i & 8 & 1+i & 1+i & 0 \\ 1-i & 1-i & 8 & 1+i & 1+i \\ 0 & 1-i & 1-i & 8 & 1+i \\ 0 & 0 & 1-i & 1-i & 8 \end{bmatrix}$$

The following program fragments stores H in h, using band symmetric storage format:

```
h = complexarr(15)
h(0:1) = 0
h(2:4) = complex(1,1)
h(5) = 0
h(6:9) = complex(1,1)
h(10:14) = 8
```

Choosing Between Banded and Coordinate Forms

It is clear that any matrix can be stored in either sparse coordinate or band format. The choice depends on the sparsity pattern of the matrix. A matrix with all non-zero data stored in bands close to the main diagonal would probably be a

good candidate for band format. If non-zero information is scattered more or less uniformly through the matrix, sparse coordinate format is the best choice. As extreme examples, consider the following two cases. First, consider an n x n matrix with all elements on the main diagonal and the `(0,n-1)` and `(n-1,0)` entries non-zero. The sparse coordinate vector would be $n + 2$ units long. An array of length $2n^2 - n$ would be required to store the band representation, nearly twice as much storage as a dense solver might require. Secondly, consider a tridiagonal matrix with all diagonal, superdiagonal and subdiagonal entries non-zero. In band format, an array of length $3n$ is needed. In sparse coordinate format, a vector of length $3n - 2$ is required. But the problem is that, for instance with floating-point precision on a 32 bit machine, each of those $3n - 2$ units in coordinate format requires three times as much storage as any of the $3n$ units needed for band representation. This is due to carrying the row and column indices in coordinate form. Band storage evades this requirement by being essentially an ordered list, and defining location in the original matrix by position in the list.

Compressed Sparse Column (CSC) Format

Functions that accept data in coordinate format can also accept data stored in the format described in the *Users' Guide for the Harwell-Boeing Sparse Matrix Collection*. The scheme is column oriented, with each column held as a sparse vector, represented by a list of the row indices of the entries in an integer array and a list of the corresponding values in a separate *float* (*double*, *complex*, *dcomplex*) array. Data for each column are stored consecutively and in order. A separate integer array holds the location of the first entry of each column and the first free location. Only entries in the lower triangle and diagonal are stored for symmetric and Hermitian matrices. All arrays are based at zero, which is in contrast to the Harwell-Boeing test suite's one-based arrays.

As in the Harwell-Boeing *Users' Guide*, we illustrate the storage scheme with the following example. The 5x5 matrix:

$$\begin{bmatrix} 1 & -3 & 0 & -1 & 0 \\ 0 & 0 & -2 & 0 & 3 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & -4 & 0 \\ 5 & 0 & -5 & 0 & 6 \end{bmatrix}$$

would be stored in the arrays colptr (location of first entry), rowind (row indices), and `values` (non-zero entries) as follows:

Subscripts	0	1	2	3	4	5	6	7	8	9	10
colptr	0	3	5	7	9	11					
rowind	0	4	2	3	0	1	4	0	3	4	1
values	1	5	2	4	-3	-2	-5	-1	-4	6	3

INV Function

Computes the inverse of a real or complex, square matrix.

Usage

result = INV(*a*)

Input Parameters

a — Two-dimensional matrix containing the matrix to be inverted.

Returned Value

result — A two-dimensional matrix containing the inverse of the matrix *A*.

Input Keywords

Double — If present and nonzero, double precision is used.

Example

```
RM, a, 3, 3
    ; Define the matrix to be inverted.
row 0: 1 3 3
row 1: 1 3 4
row 2: 1 4 4
ainv = INV(a)
    ; Call INV to perform the inversion.
PM, a
    ; Output the original matrix.
 1.00000       3.00000       3.00000
```

```
 1.00000        3.00000        4.00000
 1.00000        4.00000        4.00000

PM, ainv
    ; Output the computed inverse.
  4.00000       -0.00000       -3.00000
  0.00000       -1.00000        1.00000
 -1.00000        1.00000        0.00000
PM, a # ainv
    ; Check the results.
 1.00000        0.00000        0.00000
 0.00000        1.00000        0.00000
 0.00000        0.00000        1.00000
```

Fatal Errors

MATH_SINGULAR_MATRIX — Input matrix is singular.

LUSOL Function

Solves a general system of real or complex linear equations $Ax = b$.

Usage

result = LUSOL(*b* [, *a*])

Input Parameters

b — One-dimensional matrix containing the right-hand side.

a — Two-dimensional matrix containing the coefficient matrix. Element *A(i, j)* contains the *j*-th coefficient of the *i*-th equation.

Returned Value

result — A one-dimensional array containing the solution of the linear system $Ax = b$.

Input Keywords

Double — If present and nonzero, double precision is used.

Transpose — If present and nonzero, $A^H x = b$ is solved.

Pivot — Specifies a named variable into which the pivot sequence for the factorization, computed by the LUFAC procedure, is stored. Keywords *Pivot* and *Factor* must be used together. Keywords *Pivot* and *Condition* cannot be used together.

Output Keywords

Factor — Specifies a named variable in which the LU factorization of A, computed by the LUFAC procedure, is stored. The strictly lower-triangular part of this array contains information necessary to construct L, and the upper-triangular part contains U. Keywords *Pivot* and *Factor* must be used together. Keywords *Factor* and *Condition* cannot be used together.

Condition — Specifies a named variable into which an estimate of the L_1 condition number is stored. This keyword cannot be used with keywords *Pivot* and *Factor*.

Inverse — Specifies a named variable into which the inverse of the matrix A is stored.

Discussion

Function LUSOL solves a system of linear algebraic equations with a real or complex coefficient matrix A. Any of several related computations can be performed by using keywords. These extra tasks include solving $A^H x = b$ or computing the solution of $Ax = b$ given the LU factorization of A. The function first computes the LU factorization of A with partial pivoting such that $L^{-1}PA = U$.

The matrix U is upper-triangular, while $L^{-1}A \equiv P_{n-1} L_{n-2} P_{n-2} \ldots L_0 P_0 A \equiv U$. The factors P_i and L_i are defined by the partial pivoting. Each P_i is an interchange of row i with row $j \geq i$. Thus, P_i is defined by that value of j. Every $L_i = m_i e_i^T$ is an elementary elimination matrix. The vector m_i is zero in entries $0, \ldots, i-1$. This vector is stored as column i in the strictly lower-triangular part of the working matrix containing the decomposition information.

The factorization efficiency is based on a technique of "loop unrolling and jamming" due to Dr. Leonard J. Harding of the University of Michigan, Ann Arbor, Mich. The solution of the linear system is then found by solving two simpler

systems, $y = L^{-1} b$ and $x = U^{-1} y$. When the solution to the linear system or the inverse of the matrix is sought, an estimate of the L_1 condition number of A is computed using the same algorithm as in Dongarra et al. (1979). If the estimated condition number is greater than 1 / ε (where ε is the machine precision), a warning message is issued. This indicates that very small changes in A may produce large changes in the solution x. Function LUSOL fails if U, the upper-triangular part of the factorization, has a zero diagonal element.

Example 1: Solving a System

This example solves a system of three linear equations. This is the simplest use of the function. The equations are as follows:

$$x_0 + 3x_1 + 3x_2 = 1$$

$$x_0 + 3x_1 + 4x_2 = 4$$

$$x_0 + 4x_1 + 3x_2 = -1$$

```
RM, a, 3, 3
    ; Input a matrix containing the coefficients.
row 0: 1 3 3
row 1: 1 3 4
row 2: 1 4 3
RM, b, 3, 1
    ; Input a vector containing the right-hand side.
row 0: 1
row 1: 4
row 2: -1
x = LUSOL(b, a)
    ; Call LUSOL to compute the solution.
PM, x, Title = 'Solution'
    ; Print solution and residual.
Solution
 -2.00000
 -2.00000
 3.00000
PM, a # x - b, Title = 'Residual'
Residual
      0.00000
```

```
      0.00000
      0.00000
```

Example 2: Transpose Problem

In this example, the transpose problem $A^H x = b$ is solved.

```
RM, a, 3, 3
    ; Input the matrix containing the coefficients.
row 0: 1 3 3
row 1: 1 3 4
row 2: 1 4 3
RM, b, 3, 1
    ; Input the vector containing the right-hand side.
row 0: 1
row 1: 4
row 2: -1
x = LUSOL(b, a, /Transpose)
    ; Call LUSOL with keyword Transpose set.
PM, x, Title = 'Solution'
    ; Print the solution and the residual.
Solution
       4.00000
      -4.00000
       1.00000
PM, TRANSPOSE(a) # x - b, Title = 'Residual'
Residual
       0.00000
       0.00000
       0.00000
```

Example 3: Solving with Multiple Right-hand Sides

This example computes the solution of two systems. Only the right-hand sides differ. The matrix and first right-hand side are given in the initial example. The second right-hand side is the vector $c = [0.5, 0.3, 0.4]^T$. The factorization information is computed by procedure LUFAC and is used to compute the solutions in calls to LUSOL.

```
RM, a, 3, 3
    ; Input the coefficient matrix.
```

```
row 0: 1 3 3
row 1: 1 3 4
row 2: 1 4 3
```

```
RM, b, 3, 1
```

; Input the first right-hand side.

```
row 0: 1
row 1: 4
row 2: -1
```

```
RM, c, 3, 1
```

; Input the second right-hand side.

```
row 0: .5
row 1: .3
row 2: .4
```

```
LUFAC, a, pvt, fac
```

; Call LUFAC to factor the coefficient matrix.

```
x = LUSOL(b, Factor = fac, Pivot = pvt)
```

; Call LUSOL with the factored form of the coefficient matrix and the
; first right-hand side.

```
PM, x, Title = 'Solution'
```

; Print the solution of $Ax = b$.

```
Solution
      -2.00000
      -2.00000
       3.00000
```

```
PM, a # x - b, Title = 'Residual'
```

```
Residual
       0.00000
       0.00000
       0.00000
```

```
y = LUSOL(c, Factor = fac, Pivot = pvt)
```

; Call LUSOL with the factored form of the coefficient matrix and the
; second right-hand side.

```
PM, y, Title = 'Solution'
```

; Print the solution of $Ax = b$.

```
Solution
      1.40000
     -0.100000
     -0.200000
```

```
PM, a # y - c, $
   Title = 'Residual', Format = '(f8.5)'

Residual
   0.00000
   0.00000
   0.00000
```

Warning Errors

MATH_ILL_CONDITIONED — Input matrix is too ill-conditioned. An estimate of the reciprocal of its L_1 condition number is #. The solution might not be accurate.

Fatal Errors

MATH_SINGULAR_MATRIX — Input matrix is singular.

LUFAC Procedure

Computes the *LU* factorization of a real or complex matrix.

Usage

LUFAC, *a* [, *pivot* [, *fac*]]

Input Parameters

a — Two-dimensional matrix containing the coefficient matrix. Element *A (i, j)* contains the *j*-th coefficient of the *i*-th equation.

Output Parameters

pivot — One-dimensional matrix containing the pivot sequence of the factorization.

fac — Two-dimensional matrix containing the *LU* factorization of *A*. The strictly lower-triangular part of this matrix contains information necessary to construct *L*, and the upper-triangular part contains *U*.

Input Keywords

Double — If present and nonzero, double precision is used.

Transpose — If present and nonzero, $A^{\mathrm{T}}X=b$ is solved.

Output Keywords

Condition — Specifies a named variable into which an estimate of the L_1 condition number is stored.

L — Specifies a named variable into which the strictly lower-triangular matrix L of the LU factorization is stored.

U — Specifies a named variable into which the upper-triangular matrix U of the LU factorization is stored.

PA — Specifies a named variable into which the matrix resulting from applying the pivot permutation to A is stored.

Inverse — Specifies a named variable into which the inverse of the matrix A is stored.

Discussion

Any of several related computations can be performed by using keywords. These extra tasks include computing the LU factorization of A^T, computing an estimate of the L_1 condition number, and returning L or U separately.

The LUFAC procedure computes the LU factorization of A with partial pivoting such that $L^{-1}PA = U$. The matrix U is upper-triangular, while $L^{-1}A \equiv P_{n-1}\,L_{n-2}P_{n-2}...L_0\,P_0\,A \equiv U$. The factors P_i and L_i are defined by the partial pivoting. Each P_i is an interchange of row i with row $i \geq j$. Thus, P_i is defined by that value of j. Every $L_i = m_i e_i^T$ is an elementary elimination matrix. The vector m_i is zero in entries 0, ..., $i - 1$. This vector is stored as column i in the strictly lower-triangular part of the working array containing the decomposition information.

The factorization efficiency is based on a technique of “loop unrolling and jamming” due to Dr. Leonard J. Harding of the University of Michigan, Ann Arbor, Mich. When the inverse of the matrix is sought, an estimate of the L_1 condition number of A is computed using the same algorithm as in Dongarra et al. (1979). If the estimated condition number is greater than $1 / \varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes

in A may produce large changes in the solution x. The LUFAC procedure fails if U, the upper-triangular part of the factorization, has a zero diagonal element.

Example 1

This example computes the LU factorization of a matrix and prints it out in the default form with the information needed to construct L and U combined in one array. The matrix is as follows:

$$\begin{bmatrix} 1 & 3 & 3 \\ 1 & 3 & 4 \\ 1 & 4 & 3 \end{bmatrix}$$

```
RM, a, 3, 3
    ; Input the matrix to be factored.
row 0: 1 3 3
row 1: 1 3 4
row 2: 1 4 3
LUFAC, a, pvt, fac
    ; Factor the matrix by calling LUFAC.
PM, fac, Title = 'LU factors of A'
    ; Print the results.
LU factors of A
       1.00000        3.00000        3.00000
      -1.00000        1.00000        0.00000
      -1.00000       -0.00000        1.00000
PM, pvt, Title = 'Pivot sequence'
Pivot sequence
             1
             3
             3
```

Example 2

This example computes the factorization, uses keywords to return the factorization in separate named variables, and returns the original matrix after the pivot permutation is applied.

```
RM, a, 3, 3
```

```
    ; Input the matrix to be factored.
row 0: 1 3 3
row 1: 1 3 4
row 2: 1 4 3
LUFAC, a, L = l, U = u, PA = pa
    ; Call LUFAC with the keywords L and U.
PM, l, Title = 'L'
    ; Print the results.
L
        1.00000        0.00000        0.00000
        1.00000        1.00000        0.00000
        1.00000        0.00000        1.00000
PM, u, Title = 'U'


U
        1.00000        3.00000        3.00000
        0.00000        1.00000        0.00000
        0.00000        0.00000        1.00000
PM, l # u - pa, $
   Title = 'Residual: L # U - PA'
Residual: L # U - PA
        0.00000        0.00000        0.00000
        0.00000        0.00000        0.00000
        0.00000        0.00000        0.00000
```

Warning Errors

MATH_ILL_CONDITIONED — Input matrix is too ill-conditioned. An estimate of the reciprocal of its L_1 condition number is #. The solution might not be accurate.

Fatal Errors

MATH_SINGULAR_MATRIX — Input matrix is singular.

CHSOL Function

Solves a symmetric positive definite system of real or complex linear equations $Ax = b$.

Usage

result = CHSOL(*b* [, *a*])

Input Parameters

b — One-dimensional matrix containing the right-hand side.

a — Two-dimensional matrix containing the coefficient matrix. Matrix *A (i, j)* contains the *j*-th coefficient of the *i*-th equation.

Returned Value

result — The solution of the linear system $Ax = b$.

Input Keywords

Double — If present and nonzero, double precision is used.

Output Keywords

Factor — Specifies a named variable in which the LL^H factorization of *A* is stored. The lower-triangular part of this matrix contains *L*, and the upper-triangular part contains L^H. Keywords *Condition* and *Factor* cannot be used together.

Condition — Specifies a named variable into which an estimate of the L_1 condition number is stored. This keyword cannot be used with *Factor*. Keywords *Condition* and *Factor* cannot be used together.

Inverse — Specifies a named variable into which the inverse of the matrix *A* is stored. This keyword is not allowed if *A* is complex.

Discussion

Function CHSOL solves a system of linear algebraic equations having a symmetric positive definite coefficient matrix *A*. The function first computes the Cholesky factorization LL^H of *A*. The solution of the linear system is then found

by solving the two simpler systems, $y = L^{-1}b$ and $x = L^{-H}y$. An estimate of the L_1 condition number of A is computed using the same algorithm as in Dongarra et al. (1979). If the estimated condition number is greater than 1 / ε (where ε is the machine precision), a warning message is issued. This indicates that very small changes in A may produce large changes in the solution x.

Function CHSOL fails if L, the lower-triangular matrix in the factorization, has a zero diagonal element.

Example 1

```
RM, a, 3, 3
    ; Define the coefficient matrix.
row 0:   1   -3   2
row 1: -3   10  -5
row 2:   2   -5   6
RM, b, 3, 1
    ; Define the right-hand side.
row 0:  27
row 1: -78
row 2:  64
x = CHSOL(b, a)
    ; Call CHSOL to compute the solution.
PM, x, Title = 'Solution'
Solution
       1.00000
      -4.00000
       7.00000
PM, a # x - b, Title = 'Residual'
Residual
       0.00000
       0.00000
       0.00000
```

Example 2

In this example, a system of five linear equations with Hermitian positive definite coefficient matrix is solved. The equations are as follows:

$$2x_0 + (-1 + i)\,x_1 = 1 + 5i$$

$$(-1 - i)\,x_0 + 4x_1 + (1 + 2i)\,x_2 = 12 - 6i$$

$$(-1 - 2i)\,x_1 + 10x_2 + 4ix_3 = 1 + (-16i)$$

$$(-4ix_2) + 6x_3 + (i + 1)x_4 = -3 - 3i$$

$$(1 - i)\,x_3 + 9x_4 = 25 + 16i$$

```
RM, a, 5, 5, /Complex
```

; Input the complex matrix A.

```
row 0: 2         (-1,1) 0        0        0
row 1: (-1,-1) 4        (1,2)   0        0
row 2: 0         (1,-2) 10       (0,4)   0
row 3: 0         0        (0,-4) 6        (1,1)
row 4: 0         0        0        (1,-1) 9
```

```
RM, b, 5, 1, /Complex
```

; Input the right-hand side.

```
row 0: (1, 5)
row 1: (12, -6)
row 2: (1, -16)
row 3: (-3, -3)
row 4: (25, 16)
```

```
x = CHSOL(b, a)
```

; Compute the solution.

```
PM, x, Title = 'Solution', Format = '("(",f8.5,",",f8.5,")")'
```

; Output the results.

```
Solution
( 2.00000, 1.00000)
( 3.00000,-0.00000)
(-1.00000,-1.00000)
( 0.00000,-2.00000)
( 3.00000, 2.00000)
```

```
PM, a # x - b, Title = 'Residual', Format =
   '("(",f8.5,",",f8.5,")")'
```

```
Residual
( 0.00000, 0.00000)
( 0.00000,-0.00000)
( 0.00000, 0.00000)
( 0.00000, 0.00000)
```

```
( 0.00000, 0.00000)
```

Warning Errors

MATH_ILL_CONDITIONED — Input matrix is too ill-conditioned. An estimate of the reciprocal of its L_1 condition number is #. The solution might not be accurate.

Fatal Errors

MATH_NONPOSITIVE_MATRIX — Leading # by # submatrix of the input matrix is not positive definite.

MATH_SINGULAR_MATRIX — Input matrix is singular.

MATH_SINGULAR_TRI_MATRIX — Input triangular matrix is singular. The index of the first zero diagonal element is #.

CHFAC Procedure

Computes the Cholesky factor, *L*, of a real or complex symmetric positive definite matrix *A*, such that $A = LL^H$.

Usage

CHFAC, *a*, *fac*

Input Parameters

a — Two-dimensional matrix containing the coefficient matrix. Element *A (i, j)* contains the *j*-th coefficient of the *i*-th equation.

Output Parameters

fac — Two-dimensional matrix containing the Cholesky factorization of *A*. Note that *fac* contains *L* in the lower triangle and L^H in the upper triangle.

Input Keywords

Double — If present and nonzero, double precision is used.

Output Keywords

Condition — Specifies a named variable into which an estimate of the L_1 condition number is stored.

Inverse — Specifies a named variable into which the inverse of the matrix A is stored. This keyword is not allowed if A is complex.

Discussion

Procedure CHFAC computes the Cholesky factorization LL^H of a symmetric positive definite matrix A. When the inverse of the matrix is sought, an estimate of the L_1 condition number of A is computed using the same algorithm as in Dongarra et al. (1979). If the estimated condition number is greater than $1 / \varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in A may produce large changes in the solution x.

The CHFAC function fails if L, the lower-triangular matrix in the factorization, has a zero diagonal element.

Example

This example computes the Cholesky factorization of a 3 x 3 matrix.

```
RM, a, 3, 3
    ; Define the matrix A.
row 0:   1   -3   2
row 1: -3   10  -5
row 2:   2   -5   6
CHFAC, a, fac
    ; Call CHFAC to compute the factorization.
PM, fac, Title = 'Cholesky factor'
Cholesky factor
       1.00000      -3.00000       2.00000
      -3.00000       1.00000       1.00000
       2.00000       1.00000       1.00000
```

Warning Errors

MATH_ILL_CONDITIONED — Input matrix is too ill-conditioned. An estimate of the reciprocal of its L_1 condition number is #. The solution might not be accurate.

Fatal Errors

MATH_NONPOSITIVE_MATRIX — Leading # by # submatrix of the input matrix is not positive definite.

MATH_SINGULAR_MATRIX — Input matrix is singular.

MATH_SINGULAR_TRI_MATRIX — Input triangular matrix is singular. The index of the first zero diagonal element is #.

QRSOL Function

Solves a real linear least-squares problem $Ax = b$.

Usage

result = QRSOL(*b* [, *a*])

Input Parameters

b — Matrix containing the right-hand side.

a — (Optional) Two-dimensional matrix containing the coefficient matrix. Element *A (i, j)* contains the *j*-th coefficient of the *i*-th equation.

Returned Value

result — The solution, *x*, of the linear least-squares problem $Ax = b$.

Input Keywords

Auxqr — Specifies a named variable in which the matrix containing the scalars τ_k of the Householder transformations that define the decomposition, as computed in the procedure QRFAC, is stored. Keywords *Auxqr, Pivot,* and *Qr* must be used together.

Double — If present and nonzero, double precision is used.

Qr — Specifies a named variable in which the matrix containing the Householder transformations that define the decomposition, as computed in the procedure QRFAC, is stored. Keywords *Auxqr*, *Pivot*, and *Qr* must be used together.

Tolerance — Nonnegative tolerance used to determine the subset of columns of *A* to be included in the solution.

Default: Tolerance = SQRT(ε), where ε is machine precision

Output Keywords

Residual — Specifies a named variable in which the matrix containing the residual vector $b - Ax$ is stored.

Basis — Named variable containing an integer specifying the number of columns used in the solution. The value *Basis* = *k*, if $|r_{k,k}| < Tolerance*|r_{0,0}|$ and $|r_{i,i}| \geq Tolerance*|r_{0,0}|$ for $i = 0, 1, \ldots, k - 1$. For more information on the use of this option, see the *Discussion* section below.

Input/Output Keywords

Pivot — Specifies a named variable in which the array containing the desired variable order and usage information is stored. Keywords *Auxqr*, *Pivot*, and *Qr* must be used together.

On input, if *Pivot* (*k*) > 0, then column *k* of *A* is an initial column. If *Pivot* (*k*) = 0, then the column of *A* is a free column and can be interchanged in the column pivoting. If *Pivot* (*k*) < 0, then column *k* of *A* is a final column. If all columns are specified as initial (or final) columns, then no pivoting is performed. (The permutation matrix *P* is the identity matrix in this case.)

On output, *Pivot* (*k*) contains the index of the column of the original matrix that has been interchanged into column *k*.

Default: *Pivot* (*) = 0

NOTE If QRSOL is used to solve a problem previously factored in procedure QRFAC, the matrix specified by *Pivot* should contain the same information that parameter *pivot* of QRFAC contained upon output.

Discussion

Function QRSOL solves a system of linear least-squares problems $Ax = b$ with column pivoting. It computes a *QR* factorization of the matrix *AP*, where *P* is the permutation matrix defined by the pivoting, and computes the smallest integer *k* satisfying $|r_{k,k}| < Tolerance*|r_{0,0}|$ to the output keyword *Basis*.

Householder transformations

$Q_k = I - \tau_k u_k u_k^T$, $k = 0, ..., \min(m - 1, n) - 1$

are used to compute the factorization. The decomposition is computed in the form
$Q_{\min\,(m-1,\,n)-1} \ldots Q_0\, AP = R$, so $AP = QR$ where $Q = Q_0 \ldots Q_{\min\,(m-1,\,n)-1}$.
Since each Householder vector u_k has zeros in the first $k + 1$ entries, it is stored as part of column k of *Qr*. The upper-trapezoidal matrix R is stored in the upper-trapezoidal part of the first $\min(m, n)$ rows of *Qr*. The solution x to the least-squares problem is computed by solving the upper-triangular system of linear equations
$R\,(0{:}k,\, 0{:}k)\; y\,(0{:}k) = (Q^T b)\,(0{:}k)$ with $k = Basis - 1$. The solution is completed by setting $y\,(k{:}n - 1)$ to zero and rearranging the variables, $x = Py$.

If *Qr* and *Auxqr* are specified, then the function computes the least-squares solution to $Ax = b$ given the *QR* factorization previously defined. There are *Basis* columns used in the solution. Hence, in the case that all columns are free, x is computed as described in the default case.

Example

This example illustrates the least-squares solution of four linear equations in three unknowns by using column pivoting. The problem is equivalent to least-squares quadratic polynomial fitting to four data values. The polynomial is written as
$p(t) = x_0 + tx_1 + t^2x_2$ and the data pairs (t_i, b_i), $t_i = 2(i + 1)$, $i = 0, 1, 2, 3$. The solution to $Ax = b$ is returned by function QRSOL.

```
RM, a, 4, 3
    ; Define the coefficient matrix.
row 0:  1 2 4
row 1:  1 4 16
row 2:  1 6 36
row 3:  1 8 64
RM, b, 4, 1
    ; Define the right-hand side.
row 0:  4.999
row 1:  9.001
row 2:  12.999
row 3:  17.001
x = QRSOL(b, a)
    ; Call QRSOL.
PM, x, Title = 'Solution', Format = '(f8.5)'
```

```
    ; Output the results.
Solution
 0.99900
 2.00020
 0.00000
PM, a # x - b, Title = 'Residual', $
   Format = '(f10.7)'
Residual
 0.0004015
-0.0011997
 0.0012007
-0.0004005
```

Fatal Errors

MATH_SINGULAR_TRI_MATRIX — Input triangular matrix is singular. The index of the first zero diagonal term is #.

QRFAC Procedure

Computes the *QR* factorization of a real matrix *A*.

Usage

QRFAC, *a* [, *pivot* [, *auxqr, qr*]]

Input Parameters

a — Two-dimensional matrix containing the coefficient matrix. Element *A(i,j)* contains the *j*-th coefficient of the *i*-th equation.

Input/Output Keywords

pivot — One-dimensional matrix containing the desired variable order and usage information.

On input, if *pivot* (*k*) > 0, then column *k* of *A* is an initial column. If *pivot* (*k*) = 0, then the column of *A* is a free column and can be interchanged in the column pivoting. If *pivot* (*k*) < 0, then column *k* of *A* is a final column. If all

columns are specified as initial (or final) columns, then no pivoting is performed. (The permutation matrix *P* is the identity matrix in this case.)

Default: *pivot* (*) = 0

On output, *pivot (k)* contains the index of the column of the original matrix that has been interchanged into column *k*.

Output Parameters

auxqr — Matrix containing the scalars τ_k of the Householder transformations that define the decomposition.

qr — Matrix containing the Householder transformations that define the decomposition.

Input Keywords

Double — If present and nonzero, double precision is used.

Tolerance — Nonnegative tolerance used to determine the subset of columns of *A* to be included in the solution.

Default: Tolerance = SQRT(ε), where ε is machine precision

Output Keywords

Basis — Named variable into which an integer containing the number of columns used in the solution is stored. The value *Basis = k*, if $|r_{k,k}| < \text{Tolerance}*|r_{0,0}|$ and $|r_{i,i}| \geq \textit{Tolerance}*|r_{0,0}|$ for i = 0, 1, …, k – 1. For more information on the use of this option, see the *Discussion* section which follows.

AP — Specifies a named variable into which the product *AP* of the identity *AP = QR* is stored. This keyword is useful when attempting to compute the residual *AP – QR*.

Q — Specifies a named variable in which the two-dimensional matrix containing the orthogonal matrix of the *AP = QR* factorization is stored.

R — Specifies a named variable in which the two-dimensional matrix containing the upper-triangular matrix of the *AP = QR* decomposition is stored.

Discussion

The QRFAC procedure computes a *QR* factorization of the matrix *AP*, where *P* is the permutation matrix defined by the pivoting and computes the smallest integer *k* satisfying $|r_{k,k}| < Tolerance*|r_{0,0}|$ to the output keyword *Basis*.

Householder transformations

$Q_k = I - \tau_k u_k u_k^T$, $k = 0, ..., \min(m - 1, n) - 1$

are used to compute the factorization. The decomposition is computed in the form $Q_{\min(m-1,\,n)-1} \ldots Q_0 AP = R$, so $AP = QR$ where $Q = Q_0 \ldots Q_{\min(m-1,\,n)-1}$. Since each Householder vector u_k has zeros in the first $k + 1$ entries, it is stored as part of column *k* of *Qr*. The upper-trapezoidal matrix *R* is stored in the upper-trapezoidal part of the first min(*m*, *n*) rows of *Qr*.

When computing the factorization, the procedure computes the *QR* factorization of *AP* with *P* defined by the input *pivot* and by column pivoting among "free" columns. Before the factorization, initial columns are moved to the beginning of the array *A* and the final columns to the end. Neither initial nor final columns are permuted further during the computation. Only the free columns are moved.

Example

Using the same data as the first example given for function QRSOL, the *QR* factorization of the coefficient matrix is computed. Using keywords, the factorization is returned in the full matrices, rather than the default condensed format.

```
RM, a, 4, 3
    ; Define the coefficient matrix.

row 0:   1 2 4
row 1:   1 4 16
row 2:   1 6 36
row 3:   1 8 64

QRFAC, a, pvt, Q = q, R = r, AP = ap
    ; Call QRFAC using keywords Q, R, and AP.

PM, q, Title = 'Q', Format = '(4f12.6)'
    ; Output the results.

Q
    -0.500000     0.670820     0.500000     0.223607
    -0.500000     0.223607    -0.500000    -0.670820
    -0.500000    -0.223607    -0.500000     0.670820
    -0.500000    -0.670820     0.500000    -0.223607
```

```
PM, r, Title = 'R', Format = '(3f12.6)'
R
    -2.000000   -10.000000   -60.000000
     0.000000    -4.472136   -44.721359
     0.000000     0.000000     8.000001
     0.000000     0.000000     0.000000
PM, pvt, Title = 'Pvt'
Pvt
             1
             2
             3
PM, q # r - ap, Title = 'Residual', $
   Format = '(3f12.6)'
Residual
     0.000000     0.000000     0.000002
     0.000000     0.000000    -0.000002
     0.000000     0.000000     0.000000
     0.000000     0.000000     0.000000
```

Fatal Errors

MATH_SINGULAR_TRI_MATRIX — Input triangular matrix is singular. The index of the first zero diagonal term is #.

SVDCOMP Function

Computes the singular value decomposition (*SVD*), $A = USV^T$, of a real or complex rectangular matrix *A*. An estimate of the rank of *A* also can be computed.

Usage

result = SVDCOMP(*a*)

Input Parameters

a — Two-dimensional matrix containing the coefficient matrix. Element *A (i, j)* contains the *j*-th coefficient of the *i*-th equation.

Returned Value

result — A one-dimensional array containing the ordered singular values of *A*.

Input Keywords

Double — If present and nonzero, double precision is used.

Tol_Rank — Specifies a named variable containing the tolerance used to determine when a singular value is negligible and replaced by the value zero. If *Tol_Rank* > 0, then a singular value $s_{i,i}$ is considered negligible if $s_{i,i} \leq$ *Tol_Rank*. If *Tol_Rank* < 0, then a singular value $s_{i,i}$ is considered negligible if $s_{i,i} \leq$ *Tol_Rank* * $||A||_{infinity}$.

In this case, |*Tol_Rank*| should be an estimate of relative error or uncertainty in the data.

Output Keywords

Rank — Specifies a named variable into which an estimate of the rank of *A* is stored.

U — Specifies a named variable into which the left-singular vectors of *A* are stored.

V — Specifies a named variable into which the right-singular vectors of *A* are stored.

Inverse — Specifies a named variable into which the generalized inverse of the matrix *A* is stored.

Discussion

Function SVDCOMP computes the singular value decomposition of a real or complex matrix *A*. It first reduces the matrix *A* to a bidiagonal matrix *B* by pre- and post-multiplying Householder transformations. Then, the singular value decomposition of *B* is computed using the implicit-shifted *QR* algorithm. An estimate of the rank of the matrix *A* is obtained by finding the smallest integer *k* such that $s_{k,k} \leq Tol\_Rank$ or $s_{k,k} \leq Tol\_Rank * ||A||_{infinity}$.

Since $s_{i+1,\,i+1} \leq s_{i,i}$, it follows that all the $s_{i,i}$ satisfy the same inequality for $i = k, ..., \min(m, n) - 2$. The rank is set to the value *k*. If $A = USV^T$, its generalized inverse is $A^+ = VS^+U^T$. Here, $S^+ = \text{diag}\,(s^{-1}_{0,0},..., s^{-1}_{i,i}, 0, ..., 0)$. Only singular values that are not negligible are reciprocated. If *Inverse* is specified, the function first computes the singular value decomposition of the matrix *A*. The generalized inverse is then computed. The SVDCOMP function fails if the *QR* algorithm does not converge after 30 iterations.

Example 1

This example computes the singular values of a 6-by-4 real matrix.

```
RM, a, 6, 4
    ; Define the matrix.
row 0: 1 2 1 4
row 1: 3 2 1 3
row 2: 4 3 1 4
row 3: 2 1 3 1
row 4: 1 5 2 2
row 5: 1 2 2 3
singvals = SVDCOMP(a)
    ; Call SVDCOMP.
PM, singvals
    ; Output the results.
        11.4850
        3.26975
        2.65336
        2.08873
```

Example 2

This example computes the singular value decomposition of the 6-by-4 real matrix *A*. Matrices *U* and *V* are returned using keywords *U* and *V*.

```
RM, a, 6, 4
    ; Define the matrix.
row 0: 1 2 1 4
row 1: 3 2 1 3
row 2: 4 3 1 4
row 3: 2 1 3 1
row 4: 1 5 2 2
row 5: 1 2 2 3
singvals = SVDCOMP(a, U = u, V = v)
    ; Call SVDCOMP with keywords U and V.
PM, singvals, Title = 'Singular values', $
   Format = '(f12.6)'
    ; Output the results.
Singular values
    11.485018
     3.269752
     2.653356
     2.088730
PM, u, Title = 'Left singular vectors, U', $
   Format = '(4f12.6)'
Left singular vectors, U
    -0.380476     0.119671     0.439083    -0.565399
    -0.403754     0.345111    -0.056576     0.214776
    -0.545120     0.429265     0.051392     0.432144
    -0.264784    -0.068320    -0.883861    -0.215254
    -0.446310    -0.816828     0.141900     0.321270
    -0.354629    -0.102147    -0.004318    -0.545800
PM, v, Title = 'Right singular vectors, V', $
   Format = '(4f12.6)'
Right singular vectors, V
    -0.444294     0.555531    -0.435379     0.551754
    -0.558067    -0.654299     0.277457     0.428336
    -0.324386    -0.351361    -0.732099    -0.485129
    -0.621239     0.373931     0.444402    -0.526066
```

Warning Errors

MATH_SLOWCONVERGENT_MATRIX — Convergence cannot be reached after 30 iterations.

CHNNDSOL Function

Solves a real symmetric nonnegative definite system of linear equations $Ax = b$. Computes the solution to $Ax = b$ given the Cholesky factor.

Usage

result = CHNNDSOL(*b* [, *a*])

Input Parameters

b — Matrix containing the right-hand side.

a — (Optional) Two-dimensional matrix containing the coefficient matrix. Element $A(i, j)$ contains the j-th coefficient of the i-th equation.

Returned Value

result — A solution x of the linear system $Ax = b$.

Input Keywords

Double — If present and nonzero, double precision is used.

Factor — The LL^T factorization of A. The lower-triangular part of this matrix contains L, and the upper-triangular part contains L^T.

Tolerance — Tolerance used in determining linear dependence.

Default: *Tolerance* = 100ε, where ε is machine precision

Output Keywords

Inverse — Specifies a named variable into which the inverse of the matrix A is stored.

Discussion

Function CHNNDSOL solves a system of linear algebraic equations having a symmetric nonnegative definite (positive semidefinite) coefficient matrix. It first computes a Cholesky (LL^H or R^HR) factorization of the coefficient matrix A.

The factorization algorithm is based on the work of Healy (1968) and proceeds sequentially by columns. The i-th column is declared to be linearly dependent on the first $i - 1$ columns if

$$\left| a_{ii} - \sum_{j=0}^{i-1} r_{ji}^2 \right| \leq \varepsilon |a_{ii}|$$

where ε (specified by *Tolerance*) may be set by the user. When a linear dependence is declared, all elements in the i-th row of R (column of L) are set to zero.

Modifications due to Farebrother and Berry (1974) and Barrett and Healy (1978) for checking for matrices that are not nonnegative definite also are incorporated. The CHNNDSOL function declares A to be not nonnegative definite and issues an error message if either of the following conditions is satisfied:

1. $a_{ii} - \sum_{j=0}^{i-1} r_{ii}^2 < -\varepsilon |a_{ii}|$

2. $r = 0$ and $\left| a_{ik} - \sum_{j=0}^{i-1} r_{ji} r_{jk} \right| > \varepsilon \sqrt{a_{ii} a_{kk}},\ k > i$

Healy's (1968) algorithm and function CHNNDSOL permit the matrices A and R to occupy the same storage. Barrett and Healy (1978), in their remark, neglect this fact. The CHNNDSOL function uses

$$\sum_{j=0}^{i-1} r_{ij}^2 \ \textit{for}\ a_{ii}$$

in condition 2 above to remedy this problem.

If an inverse of the matrix A is required and the matrix is not (numerically) positive definite, then the resulting inverse is a symmetric g_2 inverse of A. For a matrix G to be a g_2 inverse of a matrix A, G must satisfy conditions 1 and 2 for

the Moore-Penrose inverse but generally fail conditions 3 and 4. The four conditions for G to be a Moore-Penrose inverse of A are as follows:

1. $AGA = A$
2. $GAG = G$
3. AG is symmetric
4. GA is symmetric

The solution of the linear system $Ax = b$ is computed by solving the factored version of the linear system $R^TRx = b$ as two successive triangular linear systems. In solving the triangular linear systems, if the elements of a row of R are all zero, the corresponding element of the solution vector is set to zero. For a detailed description of the algorithm, see Section 2 in Sallas and Lionti (1988). This routine is useful to solve normal equations in a linear least-squares problem.

Example

A solution to a system of four linear equations is obtained. Maindonald (1984, pp. 83–86, 104–105) discusses the computations for the factorization and solution to this problem.

```
RM, a, 4, 4
    ; Define the coefficient matrix.
row 0: 36 12 30  6
row 1: 12 20  2 10
row 2: 30  2 29  1
row 3:  6 10  1 14
RM, b, 4, 1
    ; Define the right-hand side.
row 0: 18
row 1: 22
row 2:  7
row 3: 20
x = CHNNDSOL(b, a)
    ; Define the right-hand side.
PM, x
    ; Output the results.
      0.166667
      0.500000
```

```
0.00000
1.00000
```

Warning Errors

MATH_INCONSISTENT_EQUATIONS_2 — Linear system of equations is inconsistent.

MATH_NOT_NONNEG_DEFINITE — Matrix *A* is not nonnegative definite.

CHNNDFAC Procedure

Computes the Cholesky factorization of the real matrix *A* such that $A = R^TR = LL^T$.

Usage

CHNNDFAC, *a*, *fac*

Input Parameters

a — Two-dimensional matrix containing the coefficient matrix. Element $A(i, j)$ contains the *j*-th coefficient of the *i*-th equation.

Output Parameters

fac — Matrix containing the LL^T factorization of *A*.

Input Keywords

Double — If present and nonzero, double precision is used.

Tolerance — Used in determining linear dependence.

Default: *Tolerance* = 100 ε, where ε is machine precision

Output Keywords

Inverse — Specifies a named variable into which the inverse of the matrix *A* is stored.

Discussion

The factorization algorithm is based on the work of Healy (1968) and proceeds sequentially by columns. The *i*-th column is declared to be linearly dependent on the first $i - 1$ columns if

$$\left| a_{ii} - \sum^{i-1} r_{ji}^2 \right| \leq \varepsilon |a_{ii}|$$

where ε (specified in *Tolerance*) may be set by the user. When a linear dependence is declared, all elements in the *i*-th row of *R* (column of *L*) are set to zero.

Modifications due to Farebrother and Berry (1974) and Barrett and Healy (1978) for checking for matrices that are not nonnegative definite also are incorporated. The CHNNDFAC procedure declares *A* to not be nonnegative definite and issues an error message if either of the following conditions is satisfied:

1. $a_{ii} - \sum_{j=0}^{i-1} r_{ii}^2 < -\varepsilon |a_{ii}|$

2. $r = 0$ and $\left| a_{ik} - \sum_{j=0}^{i-1} r_{ji} r_{jk} \right| > \varepsilon \sqrt{a_{ii} a_{kk}}, k > i$

Healy's (1968) algorithm and the CHNNDFAC procedure permit the matrices *A* and *R* to occupy the same storage. Barrett and Healy (1978) in their remark neglect this fact. Procedure CHNNDFAC uses

$$\sum_{j=0}^{i-1} r_{ij}^2 \text{ for } a_{ii}$$

in condition 2 above to remedy this problem.

If an inverse of the matrix *A* is required and the matrix is not (numerically) positive definite, then the resulting inverse is a symmetric g_2 inverse of *A*. For a matrix *G* to be a g_2 inverse of a matrix *A*, *G* must satisfy conditions 1 and 2 for the Moore-Penrose inverse but generally fail conditions 3 and 4. The four conditions for *G* to be a Moore-Penrose inverse of *A* are as follows:

1. $AGA = A$
2. $GAG = G$
3. AG is symmetric

4. *GA* is symmetric

Example

The symmetric nonnegative definite matrix in the initial example of CHNNDSOL is used to compute the factorization only in the first call to CHNNDFAC. Then, CHNNDSOL is called with both the LL^T factorization and the right-hand side vector as the input to compute a solution *x*.

```
RM, a, 4, 4
    ; Define the coefficient matrix.
row 0: 36 12 30 6
row 1: 12 20 2 10
row 2: 30 2 29 1
row 3: 6 10 1 14
CHNNDFAC, a, fac
    ; Compute the factorization.
PM, fac, Title = 'Factor', $
   Format = '(4f12.3)'
Factor
        6.000        2.000        5.000        1.000
        2.000        4.000       -2.000        2.000
        5.000       -2.000        0.000        0.000
        1.000        2.000        0.000        3.000
RM, b, 4, 1
    ; Define the right-hand side.
row 0: 18
row 1: 22
row 2: 7
row 3: 20
x = CHNNDSOL(b, Factor = fac)
    ; Compute the solution.
PM, x, Title = 'Solution'
    ; Output the solution.
Solution
       0.166667
       0.500000
       0.00000
       1.00000
```

Warning Errors

MATH_INCONSISTENT_EQUATIONS_2 — Linear system of equations is inconsistent.

MATH_NOT_NONNEG_DEFINITE — Matrix *A* is not nonnegative definite.

LINLSQ Function

Solves a linear least-squares problem with linear constraints.

Usage

result = LINLSQ(*a*, *b*, *c*, *bl*, *bu*, *contype*)

Input Parameters

a — Two-dimensional array of size nra by nca containing the coefficients of the least-squares equations, where nra is the number of least-squares equations and nca is the number of variables.

b — One-dimensional array of length nra containing the right-hand sides of the least-squares equations.

c — Two-dimensional array of size ncon by nca containing the coefficients of the constraints, where ncon is the number of constraints.

bl — One-dimensional array of length ncon containing the lower limit of the general constraints. If there is no lower limit on the *i*-th constraint, then *bl*(*i*) will not be referenced.

bu — One-dimensional array of length ncon containing the upper limit of the general constraints. If there is no upper limit on the *i*-th constraint, then *bu*(*i*) will not be referenced.

contype — One-dimensional array of length ncon indicating the type of constraints exclusive of simple bounds, where *contype*(*i*) = 0, 1, 2, 3 indicates =, ≤, ≥, and range constraints, respectively.

contype(i)	constraint
0	$\sum_{j=0}^{nca-1} c(i,j) \quad = bl(i)$
2	
3	
4	

Returned Value

result — One-dimensional array of length nca containing the approximate solution.

Input Keywords

Double — If present and nonzero, double precision is used.

Xlb — One-dimensional array of length nca containing the lower bound on the variables. If there is no lower bound on the *i*-th variable, then *Xlb*(*i*) should be set to 1.0e30.

Xub — One-dimensional array of length nca containing the upper bound on the variables. If there is no upper bound on the *i*-th variable, then *Xub*(*i*) should be set to −1.0e30.

Itmax — Set the maximum number of iterations.

Default: *Itmax* = 5*max(nra, nca)

Rel_Tolerance — Relative rank determination tolerance to be used.

Default: *Rel_Tolerance* = SQRT(machine epsilon).

Abs_Tolerance — Absolute rank determination tolerance to be used.

Default: *Abs_Tolerance* = SQRT(machine epsilon).

Output Keywords

Residual — Named variable into which an one-dimensional array containing the residuals $b - Ax$ of the least-squares equations at the approximate solution is stored.

Discussion

The function LINLSQ solves linear least-squares problems with linear constraints. These are systems of least-squares equations of the form

$$Ax \cong b$$

subject to

$$b_l \le Cx \le b_u$$

$$x_l \le x \le x_u$$

Here A is the coefficient matrix of the least-squares equations, b is the right-hand side, and C is the coefficient matrix of the constraints. The vectors b_l, b_u, x_l *and* x_u are the lower and upper bounds on the constraints and the variables, respectively. The system is solved by defining dependent variables $y \equiv Cx$ and then solving the least-squares system with the lower and upper bounds on x and y. The equation $Cx - y = 0$ is a set of equality constraints. These constraints are realized by heavy weighting, i.e., a penalty method, Hanson (1986, pp. 826-834).

Example 1

In this example, the following problem is solved in the least-squares sense:

$$3x_1 + 2x_2 + x_3 = 3.3$$

$$4x_1 + 2x_2 + x_3 = 2.2$$

$$2x_1 + 2x_2 + x_3 = 1.3$$

$$x_1 + x_2 + x_3 = 1.0$$

Subject to

$$x_1 + x_2 + x_3 \leq 1$$

$$0 \leq x_1 \leq 0.5$$

$$0 \leq x_2 \leq 0.5$$

$$0 \leq x_3 \leq 0.5$$

```
a   =   TRANSPOSE([[3.0, 2.0, 1.0], [4.0, 2.0, 1.0], $
                   [2.0, 2.0, 1.0], [1.0, 1.0, 1.0]])
b   =   [3.3, 2.3, 1.3, 1.0]
c   =   [[1.0], [1.0], [1.0]]
xub   =   [0.5, 0.5, 0.5]
xlb   =   [0.0, 0.0, 0.0]
contype   =   [1]
bc   =   [1.0]
    ; Note that only upper bound is set for contype =1.
sol   =   LINLSQ(b, a, c, bc, bc, contype, Xlb = xlb, Xub = xub)
PM, sol, Title = "Solution"
      0.500000
      0.300000
      0.200000
```

Example 2

The same problem solved in the first example is solved again. This time residuals of the least-squares equations at the approximate solution are returned, and the norm of the residual vector is printed.

```
a   =   TRANSPOSE([[3.0, 2.0, 1.0], [4.0, 2.0, 1.0], $
                   [2.0, 2.0, 1.0], [1.0, 1.0, 1.0]])
b   =   [3.3, 2.3, 1.3, 1.0]
c   =   [[1.0], [1.0], [1.0]]
xub   =   [0.5, 0.5, 0.5]
xlb   =   [0.0, 0.0, 0.0]
contype   =   [1]
bc   =   [1.0]
sol   =   LINLSQ(b, a, c, bc, bc, contype, Xlb = xlb, $
                  Xub = xub, Residual = residual)
```

```
PM, sol, Title = "Solution"
Solution
     0.500000
     0.300000
     0.200000
PM, residual, Title = "Residual"
Residual
     -1.00000
     0.500000
     0.500000
      0.00000
PRINT, "Norm of Residual =", NORM(residual)
Norm of Residual =       1.22474
```

SP_LUSOL Function

Solves a sparse system of linear equations $Ax = b$. Using keywords, any of several related computations can be performed.

Usage

result = SP_LUSOL(*b* [, *a*])

Input Parameters

b — One-dimensional matrix containing the right-hand side.

a — (Optional) Sparse matrix stored as an array of structures containing the coefficient matrix *A(i,j)*. See the chapter introduction for a description of structures used for sparse matrices.

Returned Value

result — A one-dimensional array containing the solution of the linear system $Ax = b$.

Input Keywords

Transpose — If present and nonzero, $A^T x = b$ is solved.

Pivoting — Scalar value specifying the pivoting method to use.
For Row Markowitz, set *Pivoting* to 1; for Column Markowitz, set *Pivoting* to 2; and for Symmetric Markowitz, set *Pivoting* to 3.
Default: *Pivoting* = 3

N_search_rows — The number of rows which have the least number of non-zero elements that will be searched for a pivot element.
Default: *N_search_rows* = 3

Iter_refine — If present and nonzero, iterative refinement will be applied.

Tol_drop — Possible fill-in is checked against this tolerance. If the absolute value of the new element is less than *Tol_drop*, it will be discarded.
Default: *Tol_drop* = 0.0

Stability — The absolute value of the pivot element must be bigger than the largest element in absolute value in its row divided by *Stability*.
Default: *Stability* = 10.0

Gwth_lim — The computation stops if the growth factor exceeds *Gwth_limit*.
Default: *Gwth_limit* = 1.0e16

Memory_block — Supply the number of non-zeros which will be added to the factor if current allocations are inadequate.
Default: *Memory_block* = N_ELEMENTS(*a*)

Hybrid_density — Enable the function to switch to a dense factorization method when the density of the active submatrix reaches $0.0 \leq$ *Hybrid_density* ≤ 1.0 and the order of the active submatrix is less than or equal to *Hybrid_order*. The keywords *Hybrid_density* and *Hybrid_order* must be used together.

Hybrid_order — Enable the function to switch to a dense factorization method when the density of the active submatrix reaches $0.0 \leq$ *Hybrid_density* ≤ 1.0 and the order of the active submatrix is less than or equal to *Hybrid_order*. The keywords *Hybrid_density* and *Hybrid_order* must be used together.

Csc_col — Accept the coefficient matrix in compressed sparse column (CSC) format. See the chapter introduction for a discussion of this storage scheme The keywords *Csc_col*, *Csc_row*, and *Csc_val* must be used together.

Csc_row — Accept the coefficient matrix in compressed sparse column (CSC) format. See the chapter introduction for a discussion of this storage scheme The keywords *Csc_col*, *Csc_row*, and *Csc_val* must be used together.

Csc_val — Accept the coefficient matrix in compressed sparse column (CSC) format. See the chapter introduction for a discussion of this storage scheme The keywords *Csc_col*, *Csc_row*, and *Csc_val* must be used together.

Factor_coord — The *LU* factorization of *A* as computed by SP_LUFAC. If this keyword is used, then the paramter *a* should not be used. This keyword is useful if solutions to systems involving the same coefficient matrix and multiple right-hand sides will be solved. Keywords *Factor_Coord* and *Condition* cannot be used together.

Output Keywords

Condition — Specifies a named variable into which an estimate of the L_1 condition number is stored. Keywords *Factor_Coord* and *Condition* cannot be used together.

Gwth_factor — Specifies a named variable into which the largest element in absolute value at any stage of the Gaussian elimination divided by the largest element in absolute value in *A* is stored.

Smallest_pvt — Specifies a named variable into which the value of the pivot element of smallest magnitude that occurred during the factorization is stored.

N_nonzero — Specifies a named variable into which the total number of nonzeros in the factor is stored.

Discussion

The function SP_LUSOL solves a system of linear equations $Ax = b$, where *A* is sparse. In its default usage, it solves the so-called *one off* problem, by first performing an *LU* factorization of *A* using the improved generalized symmetric Markowitz pivoting scheme. The factor *L* is not stored explicitly because the `saxpy` operations performed during the elimination are extended to the right-hand side, along with any row interchanges. Thus, the system $Ly = b$ is solved implicitly. The factor *U* is then passed to a triangular solver which computes the solution *x* from $Ux = y$.

If a sequence of systems $Ax = b$ are to be solved where *A* is unchanged, it is usually more efficient to compute the factorization once, and perform multiple forward and back solves with the various right-hand sides. In this case the factor *L* is explicitly stored and a record of all row as well as column interchanges is made. The solve step then solves the two triangular systems $Ly = b$ and $Ux = y$. In this case the user should first call SP_LUFAC to compute the factorization, then use the keyword *Factor_coord* with the function SP_LUSOL.

If the solution to $A^Tx = b$ is required, specify the keyword *Transpose*. This keyword only alters the forward elimination and back substitution so that the operations $U^Ty = b$ and $L^Tx = y$ are performed to obtain the solution. So, with one call to SP_LUFAC to produce the factorization, solutions to both $Ax = b$ and $A^Tx = b$ can be obtained.

The keyword *Condition* is used to calculate and return an estimation of the L_1 condition number of A. The algorithm used is due to Higham. Specification of *Condition* causes a complete L to be computed and stored, even if a one off problem is being solved. This is due to the fact that Higham's method requires a solution to problems of the form $Az = r$ and $A^Tz = r$.

The default pivoting strategy is symmetric Markowitz (*Pivoting* = 3). If a row or column oriented problem is encountered, there may be some reduction in fill-in by selecting either *Pivoting* = 1 for Row Markowitz, or *Pivoting* = 2 for column Markowitz. The Markowitz strategy will search a pre-elected number of rows or columns for pivot candidates. The default number is three, but this can be changed by using the keyword *N_search_rows*.

The keyword *Tol_drop* can be used to set a tolerance which can reduce fill-in. This works by preventing any new fill element which has magnitude less than the specified drop tolerance from being added to the factorization. Since this can introduce substantial error into the factorization, it is recommended that the keyword *Iter_refine* be used to recover more accuracy in the final solution. The trade-off is between space savings from the drop tolerance and the extra time needed in repeated solve steps needed for refinement.

The function SP_LUSOL provides the option of switching to a dense factorization method at some point during the decomposition. This option is enabled by specifying the keywords *Hybrid_density* and *Hybrid_order*. *Hybrid_density* specifies a minimum density for the active submatrix before a format switch will occur. A density of 1.0 indicates complete fill-in. *Hybrid_order* places an upper bound of the order of the active submatrix which will be converted to dense format. This is used to prevent a switch from occurring too early, possibly when the $O(n^3)$ nature of the dense factorization will cause performance degradation. Note that this option can significantly increase heap storage requirements.

Example 1

As an example, consider the following matrix:

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & -3 & -1 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 \\ -2 & 0 & 0 & 10 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Let $x^T = (1, 2, 3, 4, 5, 6)$ so that $Ax = (10, 7, 45, 33, –34, 31)^T$. The number of nonzeros in A is 15.

```
A = replicate(!F_sp_elem, 15)
```

; Define the sparse matrix A using coordinate storage format.

```
a(*).row = [0, 1, 1, 1, 2, $
            3, 3, 3, 4, 4, $
            4, 4, 5, 5, 5]

a(*).col = [0, 1, 2, 3, 2, $
            0, 3, 4, 0, 3, $
            4, 5, 0, 1, 5]

a(*).val = [10, 10, -3, -1, 15, $
       -2, 10, -1, -1, -5, $
        1, -3, -1, -2, 6]

b = [10, 7, 45, 33, -34, 31]
```

; Define the right-hand side.

```
x = SP_LUSOL(b, a)
```

; Call SP_LUSOL, then print out result and the residual.

```
PM, x

   1.0000000
   2.0000000
   3.0000000
   4.0000000
   5.0000000
   6.0000000

PM, SP_MVMUL(6, 6, a, x) - b

   0.0000000
   -8.8817842e-16
```

```
0.0000000
0.0000000
0.0000000
0.0000000
```

See Also

SP_LUFAC

SP_LUFAC Function

Computes an *LU* factorization of a sparse matrix stored in either coordinate format or CSC format. Using keywords, any of several related computations can be performed.

Usage

result = SP_LUFAC(*a*, *n_rows*)

Input Parameters

a — Sparse matrix stored as an array of structures containing the coefficient matrix *A(i,j)*. See the chapter introduction for a description of structures used for sparse matrices.

n_rows — The number of rows in *a*.

Returned Value

result — Structure containing the *LU* factorization of *A*.

Input Keywords

Transpose — If present and nonzero, $A^T x = b$ is solved.

Pivoting — Scalar value specifying the pivoting method to use.
For Row Markowitz, set *Pivoting* to 1; for Column Markowitz, set *Pivoting* to 2; and for Symmetric Markowitz, set *Pivoting* to 3.
Default: *Pivoting* = 3

N_search_rows — The number of rows which have the least number of non-zero elements that will be searched for a pivot element.
Default: *N_search_rows* = 3

Iter_refine — If present and nonzero, iterative refinement will be applied.

Tol_drop — Possible fill-in is checked against this tolerance. If the absolute value of the new element is less than *Tol_drop*, it will be discarded.
Default: *Tol_drop* = 0.0

Stability — The absolute value of the pivot element must be bigger than the largest element in absolute value in its row divided by *Stability*.
Default: *Stability* = 10.0

Gwth_lim — The computation stops if the growth factor exceeds *Gwth_limit*.
Default: *Gwth_limit* = 1.0e16

Memory_block — Supply the number of non-zeros which will be added to the factor if current allocations are inadequate.
Default: *Memory_block* = N_ELEMENTS(*a*)

Hybrid_density — Enable the function to switch to a dense factorization method when the density of the active submatrix reaches $0.0 \leq$ *Hybrid_density* ≤ 1.0 and the order of the active submatrix is less than or equal to *Hybrid_order*. The keywords *Hybrid_density* and *Hybrid_order* must be used together.

Hybrid_order — Enable the function to switch to a dense factorization method when the density of the active submatrix reaches $0.0 \leq$ *Hybrid_density* ≤ 1.0 and the order of the active submatrix is less than or equal to *Hybrid_order*. The keywords *Hybrid_density* and *Hybrid_order* must be used together.

Csc_col — Accept the coefficient matrix in compressed sparse column (CSC) format. See the chapter introduction for a discussion of this storage scheme The keywords *Csc_col*, *Csc_row*, and *Csc_val* must be used together.

Csc_row — Accept the coefficient matrix in compressed sparse column (CSC) format. See the chapter introduction for a discussion of this storage scheme The keywords *Csc_col*, *Csc_row*, and *Csc_val* must be used together.

Csc_val — Accept the coefficient matrix in compressed sparse column (CSC) format. See the chapter introduction for a discussion of this storage scheme The keywords *Csc_col*, *Csc_row*, and *Csc_val* must be used together.

Output Keywords

Condition — Specifies a named variable into which an estimate of the L_1 condition number is stored.

Gwth_factor — Specifies a named variable into which the largest element in absolute value at any stage of the Gaussian elimination divided by the largest element in absolute value in A is stored.

Smallest_pvt — Specifies a named variable into which the value of the pivot element of smallest magnitude that occurred during the factorization is stored.

N_nonzeros — Specifies a named variable into which the total number of nonzeros in the factor is stored.

Discussion

The function SP_LUFAC computes an LU factorization of A using the improved generalized symmetric Markowitz pivoting scheme.

If a sequence of systems $Ax = b$ are to be solved where A is unchanged, it is usually more efficient to compute the factorization once, and perform multiple forward and back solves with the various right-hand sides. In this case, the factor L is explicitly stored and a record of all rows as well as column interchanges is made. The solve step then solves the two triangular systems $Ly = b$ and $Ux = y$. In this case, first call SP_LUFAC to compute the factorization, then use the keyword *Factor_coord* with the function SP_LUSOL.

If the solution to $A^Tx = b$ is required, specify the keyword *Transpose*. This keyword only alters the forward elimination and back substitution so that the operations $U^Ty = b$ and $L^Tx = y$ are performed to obtain the solution. So, with one call to SP_LUFAC to produce the factorization, solutions to both $Ax = b$ and $A^Tx = b$ can be obtained.

The keyword *Condition* is used to calculate and return an estimation of the L_1 condition number of A. The algorithm used is due to Higham. Specification of *Condition* causes a complete L to be computed and stored, even if a one off problem is being solved. This is due to the fact that Higham's method requires solution to problems of the form $Az = r$ and $A^Tz = r$.

The default pivoting strategy is symmetric Markowitz (*Pivoting* = 3). If a row or column oriented problem is encountered, there may be some reduction in fill-in by selecting either *Pivoting* = 1 for Row Markowitz, or *Pivoting* = 2 for column Markowitz. The Markowitz strategy will search a pre-elected number of

rows or columns for pivot candidates. The default number is three, but this can be changed by using the keyword *N_search_rows*.

The keyword *Tol_drop* can be used to set a tolerance which can reduce fill-in. This works by preventing any new fill element which has magnitude less than the specified drop tolerance from being added to the factorization. Since this can introduce substantial error into the factorization, it is recommended that the keyword *Iter_refine* be used to recover more accuracy in the final solution. The trade-off is between space savings from the drop tolerance and the extra time needed in repeated solve steps needed for refinement.

The function SP_LUFAC provides the option of switching to a dense factorization method at some point during the decomposition. This option is enabled by specifying the keywords *Hybrid_density* and *Hybrid_order. Hybrid_density* specifies a minimum density for the active submatrix before a format switch will occur. A density of 1.0 indicates complete fill-in. *Hybrid_order* places an upper bound of the order of the active submatrix which will be converted to dense format. This is used to prevent a switch from occurring too early, possibly when the $O(n^3)$ nature of the dense factorization will cause performance degradation. Note that this option can significantly increase heap storage requirements.

Example 1

As an example, consider the following matrix:

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & -3 & -1 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 \\ -2 & 0 & 0 & 10 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Let

$x_1^T = (1, 2, 3, 4, 5, 6)$ so that

$x_1 = (10, 7, 45, 33, -34, 31)^T$,

and let

$x_2^T = (5, 10, 15, 15, 10, 5)$ so that $Ax_2 = (50, 40, 225, 130, -85, 5)^T$.

In this example we factor A using SP_LUFAC, and compute solutions to the systems $Ax_1 = b_1$ and $Ax_2 = b_2$ using the computed factor as input to SP_LUSOL.

```
A = replicate(!F_sp_elem, 15)
    ; Define the sparse matrix A using coordinate storage format.

a(*).row = [0, 1, 1, 1, 2, $
            3, 3, 3, 4, 4, $
            4, 4, 5, 5, 5]

a(*).col = [0, 1, 2, 3, 2, $
            0, 3, 4, 0, 3, $
            4, 5, 0, 1, 5]

a(*).val = [10, 10, -3, -1, 15, $
       -2, 10, -1, -1, -5, $
        1, -3, -1, -2, 6]

b1 = [10, 7, 45, 33, -34, 31]

b2 = [50, 40, 225, 130, -85, 5]
    ; Define the right-hand sides.

factor = SP_LUFAC(a, 6)
    ; Compute the LU factorization.

x1 = SP_LUSOL(b1, factor_coord = factor)
    ; Call SP_LUSOL with factor and b1, then print out result and the sum
    ; of the resuduals residual.

PM, x1

   1.0000000
   2.0000000
   3.0000000
   4.0000000
   5.0000000
   6.0000000

PM, TOTAL(ABS(SP_MVMUL(6, 6, a, x1) - b1))

   8.8817842e-16

x2 = SP_LUSOL(b2, factor_coord = factor)
    ; Call SP_LUSOL with factor and b2, then print out result and the
    ; sum of the resuduals residual.

PM, x2

   5.0000000
   10.000000
   15.000000
   15.000000
   10.000000
```

```
   5.0000000
PM, TOTAL(ABS(SP_MVMUL(6, 6, a, x2) - b2))
   1.4210855e-14
```

See Also

SP_LUSOL

SP_BDSOL Function

Solves a general band system of linear equations $Ax = b$. Using keywords, any of several related computations can be performed.

Usage

result = SP_BDSOL(*b, nlca, nuca [, a]*)

Input Parameters

b — One-dimensional matrix containing the right-hand side.

nlca — Number of lower codiagonals in *a*.

nuca — Number of upper codiagonals in *a*.

a — (Optional) Array of size (*nlca* + *nuca* + 1) x n containing the n x n banded coefficient matrix in band storage mode *A(i, j)*. See the chapter introduction for a description of band storage mode.

Returned Value

result — A one-dimensional array containing the solution of the linear system $Ax = b$.

Input Keywords

Transpose — If present and nonzero, $A^T x = b$ is solved.

Blk_factor — The blocking factor. This keyword must be set no larger than 32. Default: *Blk_factor* = 1.

Pivot — One-dimensional array containing the pivot sequence. The keywords *Pivot* and *Factor* must be used together. Keywords *Pivot* and *Condition* cannot be used together.

Factor — An array of size (2**nlca* + *nuca* + 1) x N_ELEMENTS(*b*) containing the *LU* factorization of *A* with column pivoting, as returned from SP_BDFAC. The keywords *Pivot* and *Factor* must be used together. Keywords *Factor* and *Condition* cannot be used together.

Double — If present and nonzero, double precision is used.

Output Keywords

Condition — Specifies a named variable into which an estimate of the L_1 condition number is stored. This keyword cannot be used with *Pivot* and *Factor.*

Discussion

The function SP_BDSOL solves a system of linear algebraic equations with a real or complex band matrix *A*. It first computes the *LU* factorization of *A* with based on the blocked *LU* factorization algorithm given in Du Croz, et al, (1990). Level-3 BLAS invocations were replaced by in-line loops. The blocking factor *Blk_factor* has the default value of 1, but can be reset to any positive value not exceeding 32.

The solution of the linear system is then found by solving two simpler systems, $y = L^{-1}b$ and $x = U^{-1}y$. When the solution to the linear system or the inverse of the matrix is sought, an estimate of the L_1 condition number of *A* is computed using Higham's modifications to Hager's method, as given in Higham (1988). If the estimated condition number is greater than $1/\varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in *A* may produce large changes in the solution *x*. The function SP_BDSOL fails if *U*, the upper triangular part of the factorization, has a zero diagonal element.

Example 1

Consider the 1000 x 1000 banded matrix below:

$$A = \begin{bmatrix} -1 & 4 & & & & & \\ 4 & -1 & 4 & & & & \\ & 4 & -1 & . & & & \\ & & . & . & . & & \\ & & & . & -1 & 4 & \\ & & & & 4 & -1 & 4 \\ & & & & & 4 & -1 \end{bmatrix}$$

In this example we compute the solution to $Ax = b$, where b is a random vector.

```
n_rows = 1000L
nlca = 1L
nuca = 1L
a = DBLARR(n_rows*(nlca+nuca+1))
a(1:n_rows-1) = 4
a(n_rows:2*n_rows-1) = -1
a(2*n_rows:*) = 4
    ; Fill A with the values of the bands.
seed = 123L
b = RANDOMU(seed, n_rows)
    ; Compute a random right-hand side.
x = SP_BDSOL(b, nlca, nuca, a)
    ; Compute the solution using SP_BDSOL above, and output the residual below.
PM, TOTAL(ABS(SP_MVMUL(n_rows, n_rows, $
                       nlca, nuca, a, x)-b))
   1.2367884e-13
```

See Also

SP_BDFAC

SP_BDFAC Procedure

Computes the *LU* factorization of a matrix stored in band storage mode.

Usage

SP_BDFAC, *nlca, nuca, n_rows, a, pivot, factor*

Input Parameters

nlca — Number of lower codiagonals in *a*.

nuca — Number of upper codiagonals in *a*.

n_rows — Number of rows in *a*.

a — Array of size (*nlca* + *nuca* + 1) x n containing the n x n banded coefficient matrix in band storage mode *A(i,j)*. See the chapter introduction for a description of band storage mode.

Output Parameters

pivot — One dimensional array containing the pivot sequence. Keyword *Pivot* and *Condition* cannot be used together.

factor — An array of size (2**nlca* + *nuca* + 1) x *n_rows* containing the *LU* factorization of *A* with column pivoting. Keywords *Factor* and *Condition* cannot be used together.

Input Keywords

Blk_factor — The blocking factor. This keyword must be set no larger than 32. Default: *Blk_factor* = 1.

Double — If present and nonzero, double precision is used.

Output Keywords

Condition — Specifies a named variable into which an estimate of the L_1 condition number is stored. Keyword *Condition* cannot be used with parameters *pivot* or *factor.*

Discussion

The function SP_BDFAC computes the *LU* factorization of *A* with based on the blocked *LU* factorization algorithm given in Du Croz, et al, (1990). Level-3 BLAS invocations were replaced by in-line loops. The blocking factor *Blk_factor* has the default value of 1, but can be reset to any positive value not exceeding 32.

An estimate of the L_1 condition number of *A* is computed using Higham's modifications to Hager's method, as given in Higham (1988). If the estimated condition number is greater than $1/\varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in *A* may produce large changes in the solution *x*.

Example 1

Consider the 1000 x 1000 banded matrix below:

$$A = \begin{bmatrix} -1 & 4 & & & & & & \\ 4 & -1 & 4 & & & & & \\ & 4 & -1 & . & & & & \\ & & . & . & . & & & \\ & & & . & -1 & 4 & & \\ & & & & 4 & -1 & 4 & \\ & & & & & 4 & -1 & \end{bmatrix}$$

In this example we compute the solution to $Ax_1 = b_1$ and $Ax_2 = b_2$, where b_1 and b_2 are random vectors. The factorization is computed just once, using SP_BDFAC, and the solutions are computed using SP_BDSOL.

```
n_rows = 1000L
nlca = 1L
nuca = 1L
a = DBLARR(n_rows*(nlca+nuca+1))
a(1:n_rows-1) = 4
a(n_rows:2*n_rows-1) = -1
a(2*n_rows:*) = 4
```

```
    ; Fill A with the values of the bands.
seed = 123L
b1 = RANDOMU(seed, n_rows)
b2 = RANDOMU(seed, n_rows)
    ; Fill random vectors
SP_BDFAC, nlca, nuca, n_rows, a, pivot, factor
    ; Compute the factorization using SP_BDFAC.
x1 = SP_BDSOL(b1, nlca, nuca, $
              Factor = factor, Pivot = pivot)
```

; Compute the solution of $Ax_1 = b_1$ above, and output the residual below.

```
PM, TOTAL(ABS(SP_MVMUL(n_rows, n_rows, nlca, $
                       nuca, a, x1)-b1))
    1.2367884e-13
x2 = SP_BDSOL(b2, nlca, nuca, $
              Factor = factor, Pivot = pivot)
```

; Compute the solution of $Ax_2 = b_2$ above, and output the residual below.

```
PM, TOTAL(ABS(SP_MVMUL(n_rows, n_rows, nlca, $
                       nuca, a, x2)-b2))
    9.1537888e-14
```

See Also

SP_BDSOL

SP_PDSOL Function

Solves a sparse symmetric positive definite system of linear equations $Ax = b$.

Usage

result = SP_PDSOL(*b, [, a]*)

Input Parameters

b — One-dimensional matrix containing the right-hand side.

a — (Optional) Sparse matrix stored as an array of structures containing nonzeros in lower triangle of the coefficient matrix *A(i,j)*. See the chapter introduction for a description of structures used for sparse matrices.

Returned Value

result — A one-dimensional array containing the solution of the linear system $Ax = b$.

Input Keywords

Multifrontal — If present and nonzero, perform the numeric factorization using a multifrontal technique. By default a standard factorization is computed based on a sparse compressed storage scheme. Keywords *MultiFrontal* and *Factor* cannot be used together.

Factor — The factorization of *A* as computed by SP_PDFAC. If this keyword is used, then the argument *a* should not be used. This keyword is useful if solutions to systems involving the same coefficient matrix and multiple right-hand sides will be solved.

Csc_col — Accept the coefficient matrix in compressed sparse column (CSC) format. See the chapter introduction for a discussion of this storage scheme. The keywords *Csc_col*, *Csc_row*, and *Csc_val* must be used together.

Csc_row — Accept the coefficient matrix in compressed sparse column (CSC) format. See the chapter introduction for a discussion of this storage scheme. The keywords *Csc_col*, *Csc_row*, and *Csc_val* must be used together.

Csc_val — Accept the coefficient matrix in compressed sparse column (CSC) format. See the chapter introduction for a discussion of this storage scheme. The keywords *Csc_col*, *Csc_row*, and *Csc_val* must be used together.

Output Keywords

Sm_diag — The smallest diagonal element that occurred during the numeric factorization. This keyword is not valid if the keyword *Factor* is used.

Lg_diag — The largest diagonal element that occurred during the numeric factorization. This keyword is not valid if the keyword *Factor* is used.

N_nonzero — Specifies a named variable into which the total number of non-zeros in the factor is stored. This keyword is not valid if the keyword *Factor* is used.

Discussion

The function SP_PDSOL solves a system of linear algebraic equations having a sparse symmetric positive definite coefficient matrix *A*. In this function's default usage, a symbolic factorization of a permutation of the coefficient matrix is computed first. Then a numerical factorization is performed. The solution of the linear system is then found using the numeric factor.

The symbolic factorization step of the computation consists of determining a minimum degree ordering and then setting up a sparse data structure for the Cholesky factor, *L*. This step only requires the "pattern" of the sparse coefficient matrix, that is, the locations of the non-zero elements but not any of the elements themselves.

The numerical factorization can be carried out in one of two ways. By default, the standard factorization is performed based on a sparse compressed storage scheme. This is fully described in George and Liu (1981). Optionally, a multifrontal technique can be used. The multifrontal method requires more storage but will be faster in certain cases. The multifrontal factorization is based on the routines in Liu (1987). For a detailed description of this method, see Liu (1990), also Duff and Reid (1983, 1984), Ashcraft (1987), Ashcraft et al. (1987), and Liu (1986, 1989).

If an application requires that several linear systems be solved where the coefficient matrix is the same but the right-hand sides change, the function SP_PDFAC can be used to precompute the Cholesky factor. Then the keyword *Factor* can be used in SP_PDSOL to efficiently solve all subsequent systems.

Given the numeric factorization, the solution x is obtained by the following calculations:

$Ly_1 = Pb$

$L^T y_2 = y_1$

$x = P^T y_2$

The permutation information, P, is carried in the numeric factor structure.

Example 1

As an example consider the 5 x 5 coefficient matrix:

$$= \begin{bmatrix} 10 & 0 & 1 & 0 & 2 \\ 0 & 20 & 0 & 0 & 3 \\ 1 & 0 & 30 & 4 & 0 \\ 0 & 0 & 4 & 40 & 5 \\ 2 & 3 & 0 & 5 & 50 \end{bmatrix}$$

Let $x^T = (5, 4, 3, 2, 1)$ so that $Ax = (55, 83, 103, 97, 82)^T$. The number of non-zeros in the lower triangle of A is nz = 10. The sparse coordinate form for the lower triangle is given by:

row	0	1	2	2	3	3	4	4	4	4
col	0	1	0	2	2	3	0	1	3	4
val	10	20	1	30	4	40	2	3	5	50

Since this representation is not unique, an equivalent form would be:

row	3	4	4	4	0	1	2	2	3	4
col	3	0	1	3	0	1	0	2	2	4
val	40	2	3	5	10	20	1	30	4	50

```
A = REPLICATE(!F_sp_elem, 10)
a(*).row = [0, 1, 2, 2, 3, 3, 4, 4, 4, 4]
a(*).col = [0, 1, 0, 2, 2, 3, 0, 1, 3, 4]
```

```
a(*).val = [10, 20, 1, 30, 4, 40, 2, 3, 5, 50]
b = [55.0d0, 83, 103, 97, 82]
x = SP_PDSOL(b, a)
PM, x
   5.0000000
   4.0000000
   3.0000000
   2.0000000
   1.0000000
```

See Also

SP_PDFAC

SP_PDFAC Function

Computes a factorization of a sparse symmetric positive definite system of linear equations $Ax = b$.

Usage

result = SP_PDFAC(*a*, *n_rows*)

Input Parameters

a — Sparse matrix stored as an array of structures containing non-zeros in lower triangle of the coefficient matrix *A(i,j)*. See the chapter introduction for a description of structures used for sparse matrices.

n_rows — The number of rows in *a*.

Returned Value

result — The factorization of $Ax = b$.

Input Keywords

Multifrontal — If present and nonzero, perform the numeric factorization using a multifrontal technique. By default a standard factorization is computed based on a sparse compressed storage scheme

Csc_col — Accept the coefficient matrix in compressed sparse column (CSC) format. See the chapter introduction for a discussion of this storage scheme. The keywords *Csc_col*, *Csc_row*, and *Csc_val* must be used together.

Csc_row — Accept the coefficient matrix in compressed sparse column (CSC) format. See the chapter introduction for a discussion of this storage scheme. The keywords *Csc_col*, *Csc_row*, and *Csc_val* must be used together.

Csc_val — Accept the coefficient matrix in compressed sparse column (CSC) format. See the chapter introduction for a discussion of this storage scheme. The keywords *Csc_col*, *Csc_row*, and *Csc_val* must be used together.

Output Keywords

Sm_diag — The smallest diagonal element that occurred during the numeric factorization.

Lg_diag — The largest diagonal element that occurred during the numeric factorization.

N_nonzero — Specifies a named variable into which the total number of non-zeros in the factor is stored.

Discussion

The function SP_PDFAC computes a factorization of a sparse symmetric positive definite coefficient matrix *A*. In this function's default usage, a symbolic factorization of a permutation of the coefficient matrix is computed first. Then a numerical factorization is performed.

The symbolic factorization step of the computation consists of determining a minimum degree ordering and then setting up a sparse data structure for the Cholesky factor, *L*. This step only requires the "pattern" of the sparse coefficient matrix, that is, the locations of the non-zero elements but not any of the elements themselves.

The numerical factorization can be carried out in one of two ways. By default, the standard factorization is performed based on a sparse compressed storage scheme. This is fully described in George and Liu (1981). Optionally, a multifrontal technique can be used. The multifrontal method requires more storage but will be faster in certain cases. The multifrontal factorization is based on the routines in Liu (1987). For a detailed description of this method, see Liu (1990), also Duff and Reid (1983, 1984), Ashcraft (1987), Ashcraft, et al. (1987), and Liu (1986, 1989).

If an application requires that several linear systems be solved where the coefficient matrix is the same but the right-hand sides change, the function SP_PDFAC can be used to precompute the Cholesky factor. Then the keyword *Factor* can be used in SP_PDSOL to efficiently solve all subsequent systems.

Given the numeric factorization, the solution x is obtained by the following calculations:

$Ly_1 = Pb$

$L^T y_2 = y_1$

$x = P^T y_2$

The permutation information, P, is carried in the numeric factor structure.

Example 1

As an example consider the 5 x 5 coefficient matrix:

$$= \begin{bmatrix} 10 & 0 & 1 & 0 & 2 \\ 0 & 20 & 0 & 0 & 3 \\ 1 & 0 & 30 & 4 & 0 \\ 0 & 0 & 4 & 40 & 5 \\ 2 & 3 & 0 & 5 & 50 \end{bmatrix}$$

Let $x_1^T = (5, 4, 3, 2, 1)$ so that $Ax_1 = (55, 83, 103, 97, 82)^T$. Let $x_2^T = (1, 2, 3, 4, 5)$ so that $Ax_2 = (23, 55, 107, 197, 278)^T$. The number of non-zeros in the lower triangle of A is nz = 10. The sparse coordinate form for the lower triangle is given by:

row	0	1	2	2	3	3	4	4	4	4
col	0	1	0	2	2	3	0	1	3	4
val	10	20	1	30	4	40	2	3	5	50

Since this representation is not unique, an equivalent form would be:

row	3	4	4	4	0	1	2	2	3	4
col	3	0	1	3	0	1	0	2	2	4
val	40	2	3	5	10	20	1	30	4	50

```
A = REPLICATE(!F_sp_elem, 10)
a(*).row = [0, 1, 2, 2, 3, 3, 4, 4, 4, 4]
a(*).col = [0, 1, 0, 2, 2, 3, 0, 1, 3, 4]
a(*).val = [10, 20, 1, 30, 4, 40, 2, 3, 5, 50]
b1 = [55, 83, 103, 97, 82]
b2 = [23, 55, 107, 197, 278]
factor = SP_PDFAC(a, 5)
x1 = SP_PDSOL(b1, FACTOR = factor)
PM, x1
   5.0000000
   4.0000000
   3.0000000
   2.0000000
   1.0000000
x2 = SP_PDSOL(b2, FACTOR = factor)
PM, x2
   1.0000000
   2.0000000
   3.0000000
   4.0000000
   5.0000000
```

See Also

SP_PDSOL

SP_BDPDSOL Function

Solves a symmetric positive definite system of linear equations $Ax = b$ in band symmetric storage mode. Using keywords, any of several related computations can be performed.

Usage

result = SP_BDPDSOL(*b, ncoda, [, a]*)

Input Parameters

b — One-dimensional matrix containing the right-hand side.

ncoda — Number of upper codiagonals in *a*.

a — (Optional) Array of size (*ncoda* + 1) x n containing the n x n banded coefficient matrix in band symmetric storage mode *A(i, j)*. See the chapter introduction for a description of band symmetric storage mode.

Returned Value

result — A one-dimensional array containing the solution of the linear system $Ax = b$.

Input Keywords

Factor — An array of size (*ncoda* + 1) x N_ELEMENTS(*b*) containing the R^TR factorization of *A* in band symmetric storage mode, as returned from SP_BDPDFAC.

Double — If present and nonzero, double precision is used.

Output Keywords

Condition — Specifies a named variable into which an estimate of the L_1 condition number is stored. This keyword cannot be used if a previously computed factorization is specified with *Factor*.

Discussion

The function SP_BDPDSOL solves a system of linear algebraic equations with a symmetric positive definite band coefficient matrix A. It computes the R^TR Cholesky factorization of A. R is an upper triangular band matrix.

The L_1 condition number of A is computed using Higham's modifications to Hager's method, as given in Higham (1988). If the estimated condition number is greater than $1/\varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in A may produce large changes in the solution x.

The function SP_BDPDSOL fails if any submatrix of R is not positive definite or if R has a zero diagonal element. These errors occur only if A is very close to a singular matrix or to a matrix which is not positive definite.

The function SP_BDPDSOL is partially based on the LINPACK subroutines CPBFA and SPBSL; see Dongarra et al. (1979).

Example 1

Solve a system of linear equations $Ax = b$, where

$$A = \begin{bmatrix} 2 & 0 & -1 & 0 \\ 0 & 4 & 2 & 1 \\ -1 & 2 & 7 & -1 \\ 0 & 1 & -1 & 3 \end{bmatrix}, \; b = \begin{bmatrix} 6 \\ -11 \\ -11 \\ 19 \end{bmatrix}$$

```
n = 4L
ncoda = 2L
a = DBLARR((ncoda+1)*n)
a(0:n-1) = [0, 0, -1, 1]
a(n:2L*n-1) = [0, 0, 2, -1]
a(2L*n:*) = [2, 4, 7, 3]
    ; Define A in band symmetric storage mode.
b = [6, -11, -11, 19]
x = SP_BDPDSOL(b, ncoda, a)
; Compute the solution
PM, x
        4.0000000
       -6.0000000
```

```
2.0000000
9.0000000
```

SP_BDPDFAC Function

Computes the R^TR Cholesky factorization of symmetric positive definite matrix, *A*, in band symmetric storage mode.

Usage

result = SP_BDPDFAC(*a, n, ncoda*)

Input Parameters

a — Array of size (*ncoda* + 1) x *n* containing the *n* x *n* banded coefficient matrix in band symmetric storage mode *A(i,j)*. See the chapter introduction for a description of band symmetric storage mode.

n — Number rows in *a*.

ncoda — Number of upper codiagonals in *a*.

Returned Value

result — An array of size (*ncoda* + 1) x *n* containing the R^TR factorization of *A* in band symmetric storage mode.

Input Keywords

Double — If present and nonzero, double precision is used.

Output Keywords

Condition — Specifies a named variable into which an estimate of the L_1 condition number is stored.

Discussion

The function SP_BDPDFAC computes the R^TR Cholesky factorization of *A*. *R* is an upper triangular band matrix.

The L_1 condition number of A is computed using Higham's modifications to Hager's method, as given in Higham (1988). If the estimated condition number is greater than $1/\varepsilon$ (where ε is the machine precision), a warning message is issued. This indicates that very small changes in A may produce large changes in the solution x.

The function SP_BDPDFAC fails if any submatrix of R is not positive definite or if R has a zero diagonal element. These errors occur only if A is very close to a singular matrix or to a matrix which is not positive definite.

The function SP_BDPDFAC is partially based on the LINPACK subroutines CPBFA and SPBSL; see Dongarra et al. (1979).

Example 1

Solve a system of linear equations $Ax = b$, using both SP_BDPDFAC and SP_BDPDSOL, where

$$A = \begin{bmatrix} 2 & 0 & -1 & 0 \\ 0 & 4 & 2 & 1 \\ -1 & 2 & 7 & -1 \\ 0 & 1 & -1 & 3 \end{bmatrix}, \; b = \begin{bmatrix} 6 \\ -11 \\ -11 \\ 19 \end{bmatrix}$$

```
n = 4L
ncoda = 2L
a = DBLARR((ncoda+1)*n)
a(0:n-1) = [0, 0, -1, 1]
a(n:2L*n-1) = [0, 0, 2, -1]
a(2L*n:*) = [2, 4, 7, 3]
    ; Define A in band symmetric storage mode.
b = [6, -11, -11, 19]
factor = SP_BDPDFAC(a, n, ncoda)
    ; Use SP_BDPDFAC to compute the factorization.
x = SP_BDPDSOL(b, ncoda, Factor=factor)
    ; Compute the solution
PM, x
       4.0000000
      -6.0000000
       2.0000000
       9.0000000
```

SP_GMRES Function

Solves a linear system $Ax = b$ using the restarted generalized minimum residual (GMRES) method.

Usage

result = SP_GMRES(*amultp, b*)

Input Parameters

amultp — Scalar string specifying a user supplied function that computes $z = Ap$. The function accepts the argument p, and returns the vector Ap.

b — One-dimensional matrix containing the right-hand side.

Returned Value

result — A one-dimensional array containing the solution of the linear system $Ax = b$.

Input Keywords

Tolerance — The algorithm attempts to generate x such that

$$\|b - Ax\|_2 \leq \tau \|b\|_2 \ ,$$

where τ = *Tolerance*.
Default: *Tolerance* = SQRT(machine precision).

Precond — Scalar sting specifying a user supplied function which sets $z = M^{-1}r$, where M is the preconditioning matrix.

Max_krylov — The maximum Krylov subspace dimension, that is, the maximum allowable number of GMRES iterations allowed before restarting.
Default: *Max_krylov*= Minimum(N_ELEMENTS(*b*), 20).

Hh_reorth — If present and nonzero, perform orthogonalization by Householder transformations, replacing the Gram-Schmidt process.

Double — If present and nonzero, double precision is used.

Input/Output Keywords

Itmax — Initially set to the maximum number of GMRES iterations allowed. On exit, the number of iterations used is returned.
Default: *Itmax*= 1000

Discussion

The function SP_GMRES, based on the FORTRAN subroutine GMRESD by H. F. Walker, solves the linear system $Ax = b$ using the GMRES method. This method is described in detail by Saad and Schultz (1986) and Walker (1988).

The GMRES method begins with an approximate solution x_0 and an initial residual $r_0 = b - Ax_0$. At iteration m, a correction z_m is determined in the Krylov subspace

$$\kappa_m(v) = \text{span}(v, Av, ..., A^{m-1}v)$$

$v = r_0$ which solves the least squares problem

$$\min_{(z \in \kappa_m(r_0))} \|b - A(x_0 + z)\|_2$$

Then at iteration m, $x_m = x_0 + z_m$.

Orthogonalization by Householder transformations requires less storage but more arithmetic than Gram-Schmidt. However, Walker (1988) reports numerical experiments which suggest the Householder approach is more stable, especially as the limits of residual reduction are reached.

Example 1

In this example, the solution to a linear system is found. The coefficient matrix is stored in coordinate format. Consider the following matrix:

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & -3 & -1 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 \\ -2 & 0 & 0 & 10 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Let $x^T = (1, 2, 3, 4, 5, 6)$ so that $Ax = (10, 7, 45, 33, -34, 31)^T$. The number of nonzeros in A is 15.

```
FUNCTION Amultp, p
    ; This function uses SP_MVMUL to multiply a sparse matrix stored
    ; in coordinate storage mode and a vector. The common block is used
    ; to hold the sparse matrix.
    COMMON Gmres_ex1, nrows, ncols, a
    RETURN, SP_MVMUL(nrows, ncols, a, p)
END


PRO Gmres1
    ; This procedure defines the sparse matrix, A, stored in coordinate
    ; storage mode, and then calls SP_GMRES to compute the solution
    ; to Ax = b.
    COMMON Gmres_ex1, nrows, ncols, a
    A = replicate(!F_sp_elem, 15)
    a(*).row = [0, 1, 1, 1, 2, $
                3, 3, 3, 4, 4, $
                     4, 4, 5, 5, 5]
    a(*).col = [0, 1, 2, 3, 2, $
                0, 3, 4, 0, 3, $
                4, 5, 0, 1, 5]
    a(*).val = [10, 10, -3, -1, 15, $
                -2, 10, -1, -1, -5, $
                 1, -3, -1, -2, 6]
    nrows = 6
    ncols = 6
    b = [10, 7, 45, 33, -34, 31]
    itmax = 10
      ; Itmax is input/output.
    x = sp_gmres('amultp', b, Itmax = itmax)
    pm, x, title = 'Solution to Ax = b'
    pm, itmax, title = 'Number of iterations'
END
    ; Output of this procedure:
Solution to Ax = b
    1.0000000
    2.0000000
```

```
   3.0000000
   4.0000000
   5.0000000
   6.0000000

Number of iterations
        6
```

SP_CG Function

Solves a real symmetric definite linear system using a conjugate gradient method. Using keywords, a preconditioner can be supplied.

Usage

result = SP_CG(*amultp, b*)

Input Parameters

amultp — Scalar string specifying a user supplied function which computes $z = Ap$. The function accepts the argument p, and returns the vector Ap.

b — One-dimensional matrix containing the right-hand side.

Returned Value

result — A one-dimensional array containing the solution of the linear system $Ax = b$.

Input Keywords

Precond — Scalar string specifying a user supplied function which sets $z = M^{-1}r$, where M is the preconditioning matrix.

Jacobi — If present, use the Jacobi preconditioner, that is, M = diag(A). The user-supplied vector Jacobi should be set so that jacobi(i) = $A_{i,i}$.

Double — If present and nonzero, double precision is used.

Input/Output Keywords

Itmax — Initially set to the maximum number of GMRES iterations allowed. On exit, the number of iterations used is returned.
Default: *Itmax* = 1000

Rel_err — Initially set to relative error desired. On exit, the computed relative error is returned. Default: *Rel_err* = SQRT(machine precision)

Discussion

The function SP_CG solves the symmetric definite linear system $Ax = b$ using the conjugate gradient method with optional preconditioning. This method is described in detail by Golub and Van Loan (1983, chapter 10), and in Hageman and Young (1981, chapter 7).

The preconditioning matrix M, is a matrix that approximates A, and for which the linear system $Mz = r$ is easy to solve. These two properties are in conflict; balancing them is a topic of much current research. In the default usage of SP_CG, $M = I$. If the keyword *Jacobi* is selected, M is set to the diagonal of A.

The number of iterations needed depends on the matrix and the error tolerance. As a rough guide,

$$Itmax = \sqrt{n} \quad for\ n \gg 1.$$

See the academic references for details.

Let M be the preconditioning matrix, let b, p, r, x, and z be vectors and let τ be the desired relative error. Then the algorithm used is as follows:

$\lambda = -1$

$p_0 = x_0$
$r_1 = b - Ap$
for $k = 1, \ldots,$ itmax
$z_k = M^{-1} r_k$
if $k = 1$ then
$\beta_k = 1$
$p_k = z_k$
else
$\beta_k = (z_k^T r_k)/(z_{k-1}^T r_{k-1})$
$p_k = z_k + \beta_k p_k$
endif
$z_k = Ap$
$\alpha_k = (z_{k-1}^T z_{k-1})/(z_k^T p_k)$

$x_k = x_k + \alpha_k p_k$
$r_k = r_k - \alpha_k z_k$
if($\|z_k\|_2 \le \tau(1-\lambda)\|x_k\|_2$)then
recomputeλ
if($\|z_k\|_2 \le \tau(1-\lambda)\|x_k\|_2$)exit
endif
endfor

Here λ is an estimate of $\lambda_{max}(G)$, the largest eigenvalue of the iteration matrix $G = I - M^{-1}A$. The stopping criterion is based on the result (Hageman and Young, 1981, pages 148-151):

$$\frac{\|x_k - x\|_M}{\|x\|_M} \le \left(\frac{1}{1-\lambda_{max}(G)}\right)\left(\frac{\|z_k\|_M}{\|x_k\|_M}\right) \quad ,$$

where $\|x\|_M^2 = x^T M x$. It is also known that

$$\lambda_{max}(T_1) \le \lambda_{max}(T_2) \le \ldots \le \lambda_{max}(G) < 1 \, ,$$

where the T_n are the symmetric, tridiagonal matrices

$$T_n = \begin{bmatrix} \mu_1 & \omega_2 & & \\ \omega_2 & \mu_2 & \omega_3 & \\ & \omega_3 & \mu_3 & \omega_4 \\ & & & \end{bmatrix}$$

with $\mu_k = 1 - \beta_k / \alpha_{k-1}$, $\mu_1 = 1 - 1 / \alpha_1$, and $\omega_k = SQRT(\beta_k)/\alpha_{k-1}$. Usually the eigenvalue computation is needed for only a few of the iterations.

Example 1

In this example, the solution to a linear system is found. The coefficient matrix is stored as a full matrix.

```
FUNCTION Amultp, p
    ; Since A is in dense form, we use the # operator to perform the
    ; matrix-vector product. The common block us used to hold A.
    COMMON Cg_comm1, a
    RETURN, a#p
END
Pro CG_EX1
```

```
    COMMON Cg_comm1, a
    a = TRANSPOSE([[ 1, -3, 2], $
                   [-3, 10, -5], $
                   [ 2, -5, 6]])
    b = [27, -78, 64]
    x = SP_CG('amultp', b)
    ; Use SP_CG to compute the solution, then output the result.
    PM, x, title = 'Solution to Ax = b'
END
    ; Output of this procedure:
Solution to Ax = b
       1.0000000
      -4.0000000
       7.0000000
```

SP_MVMUL Function

Computes a matrix-vector product involving sparse matrix and a dense vector.

Usage

Matrix stored in coordinate format:
result = SP_MVMUL(*n_rows, n_cols, a, x*)

Matrix stored in Band format:
result = SP_MVMUL(*n_rows, n_cols, nlca, nuca, a, x*)

Input Parameters

nrows — Number of rows in the matrix *a*.

ncols — Number of columns in the matrix *a*.

nlca — Number of lower codiagonals in *a*. *nuca* should be used if *a* is stored in band format.

nuca — Number of upper codiagonals in *a*. *nlca* should be used if *a* is stored in band format.

a — If in coordinate format, a sparse matrix stored as an array of structures. If banded, an array of size (*nlca* + *nuca* + 1) x *nrows* containing the *nrows* x *ncols*

banded coefficient matrix in band storage mode. If the banded, and the keyword *Symmetric* is set, an array of size (*nlca* + 1) x *nrows* containing the *nrows* x *ncols* banded coefficient matrix in band symmetric storage mode *A(i,j)*. See the chapter introduction for a description of band storage mode.

x — One-dimensional matrix containing the vector to be multiplied by *a*.

Returned Value

result — A one-dimensional array containing the product $Ax = b$.

Input Keywords

Symmetric — If present and nonzero, then *a* is stored in symmetric mode. If *A* is in coordinate format, then $Ax + A^Tx - diag(A)$ is returned. If *A* is banded, then it must be in band symmetric storage mode. See the chapter introduction for a description of band storage modes.

Discussion

The function SP_MVMUL computes a matrix-vector product involving a sparse matrix and a dense vector.

If *A* is stored in coordinate format, then the arguments *nrows*, *ncols*, *a*, and *x* should be used. If the keyword *Symmetric* is set, then $Ax + A^Tx - diag(A)$ is returned.

If *A* is a banded, then the arguments *nrows*, *ncols*, *nlca*, *nuca*, *a*, and *x* should be used. If the keyword *Symmetric* is set, then *A* must be in band symmetric storage mode, and the number of codiagonals should be used for both *nlca* and *nuca*.

Example 1

In this example, *Ax* is computed where *A* is stored in coordinate format.

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & -3 & -1 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 \\ -2 & 0 & 0 & 10 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Let
$x^T = (1, 2, 3, 4, 5, 6)$

```
A = replicate(!F_sp_elem, 15)
    ; Define the sparse matrix A using coordinate storage format.
a(*).row = [0, 1, 1, 1, 2, $
            3, 3, 3, 4, 4, $
            4, 4, 5, 5, 5]
a(*).col = [0, 1, 2, 3, 2, $
            0, 3, 4, 0, 3, $
            4, 5, 0, 1, 5]
a(*).val = [10, 10, -3, -1, 15, $
       -2, 10, -1, -1, -5, $
        1, -3, -1, -2, 6]
x= [1, 2, 3, 4, 5, 6]]
ax = SP_MVMUL(6, 6, a, x)
PM, ax
     10.000000
     7.0000000
     45.000000
     33.000000
    -34.000000
     31.000000
```

Example 2

In this example, *Ax* is computed where *A* is stored in band mode. Consider the 1000 x 1000 banded matrix below:

$$A = \begin{bmatrix} -1 & 4 & & & & & \\ 4 & -1 & 4 & & & & \\ & 4 & -1 & . & & & \\ & & . & . & . & & \\ & & & . & -1 & 4 & \\ & & & & 4 & -1 & 4 \\ & & & & & 4 & -1 \end{bmatrix}$$

Let `x(*)` = 2.

```
n_rows = 1000L
nlca = 1L
nuca = 1L
a = DBLARR(n_rows*(nlca+nuca+1))
a(1:n_rows-1) = 4
a(n_rows:2*n_rows-1) = -1
a(2*n_rows:*) = 4
    ; Fill A with the values of the bands.
x = DBLARR(n_rows)
x(*) = 2
    ; Fill up x.
expected = DBLARR(n_rows)
expected(*) = 14
expected(0) = 6
expected(n_rows-1) = 6
    ; Define the expected result.
ax = SP_MVMUL(n_rows, n_rows, nlca, $
              nuca, a, x)
    ; Compute the product, then output the difference between the
    ; computed result and the expected result.
PRINT, TOTAL(ABS(ax-expected))
   0.0000000
```

Example 3

In this example, Ax is computed where A is stored in band symmetric mode. Let

$$A = \begin{bmatrix} 2 & 0 & -1 & 0 \\ 0 & 4 & 2 & 1 \\ -1 & 2 & 7 & -1 \\ 0 & 1 & -1 & 3 \end{bmatrix}, \quad x = \begin{bmatrix} 4 \\ -6 \\ 2 \\ 9 \end{bmatrix}$$

```
n = 4L

ncoda = 2L

a = DBLARR((ncoda+1)*n)

a(0:n-1) = [0, 0, -1, 1]

a(n:2L*n-1) = [0, 0, 2, -1]

a(2L*n:*) = [2, 4, 7, 3]
    ; Fill up contents of A.

x = [4, -6, 2, 9]

ax = SP_MVMUL(n, n, ncoda, ncoda, a, x, $
              /Symmetric)
    ; Call SP_MVMUL with the keyword Symmetric set.

PM, ax
     6.0000000
    -11.000000
    -11.000000
     19.000000
```

2

CHAPTER

Eigensystem Analysis

Contents of Chapter

Introduction

An ordinary linear eigensystem problem is represented by the equation $Ax = \lambda x$, where A denotes an $n \times n$ matrix. The value λ is an *eigenvalue*, and $x \neq 0$ is the corresponding *eigenvector*. The eigenvector is determined up to a scalar factor. In all functions, this factor has been chosen so that x has Euclidean length 1, and the component of x of largest magnitude is positive. If x is a complex vector, this component of largest magnitude is scaled to be real and positive. The entry where this component occurs can be arbitrary for eigenvectors having nonunique maximum magnitude values.

A generalized linear eigensystem problem is represented by $Ax = \lambda Bx$, where A and B are $n \times n$ matrices. The value λ is a generalized eigenvalue, and x is the corresponding generalized eigenvector. The generalized eigenvectors are normalized in the same manner as for ordinary eigensystem problems.

Error Analysis and Accuracy

This section discusses ordinary eigenvalue problems. Except in special cases, functions do not return the exact eigenvalue-eigenvector pair for the ordinary eigenvalue problem $Ax = \lambda x$. Typically, the computed pair

$$\tilde{x}, \tilde{\lambda}$$

is an exact eigenvector-eigenvalue pair for a "nearby" matrix $A + E$. Information about E is known only in terms of bounds of the form

$$\| E \|_2 \leq f(n) \| A \|_2 \varepsilon .$$

The value of $f(n)$ depends on the algorithm but is typically a small fractional power of n. The parameter ε is the machine precision. By a theorem due to Bauer and Fike (see Golub and Van Loan 1989, p. 342),

$$\min|\hat{\lambda} - \lambda| \leq \kappa(X)\|E\|_2 \quad \textit{for all } \lambda \textit{ in } \sigma(A)$$

where $\sigma(A)$ is the set of all eigenvalues of A (called the spectrum of A), X is the matrix of eigenvectors,

$$\| \cdot \|_2$$

is Euclidean length, and $\kappa(X)$ is the condition number of X defined as

$$\kappa(X) = \| X \|_2 \| X^{-1} \|_2 .$$

If A is a real symmetric or complex Hermitian matrix, then its eigenvector matrix X is respectively orthogonal or unitary. For these matrices, $\kappa(X) = 1$.

The accuracy of the computed eigenvalues

$$\tilde{\lambda}_j \textit{ and eigenvectors } \tilde{x}_j$$

can be checked by computing their performance index τ. The performance index is defined to be

$$\tau = \max_{1 \le j \le n} \frac{\left\| A\tilde{x}_j - \tilde{\lambda}_j \tilde{x}_j \right\|_2}{n\varepsilon \|A\|_2 \|\tilde{x}_j\|_2}$$

where ε is again the machine precision.

The performance index τ is related to the error analysis because

$$\left\| E\tilde{x}_j \right\|_2 = \left\| A\tilde{x}_j - \tilde{\lambda}_j \tilde{x}_j \right\|_2$$

where *E* is the "nearby" matrix discussed above.

While the exact value of τ is precision and data dependent, the performance of an eigensystem analysis function is defined as excellent if $\tau < 1$, good if $1 \le \tau \le 100$, and poor if $\tau > 100$. This is an arbitrary definition, but large values of τ can serve as a warning that there is a significant error in the calculation.

If the condition number κ (*X*) of the eigenvector matrix *X* is large, there can be large errors in the eigenvalues even if τ is small. In particular, it is often difficult to recognize near multiple eigenvalues or unstable mathematical problems from numerical results. This facet of the eigenvalue problem is often difficult for users to understand. Suppose the accuracy of an individual eigenvalue is desired. This can be answered approximately by computing the *condition number of an individual eigenvalue* (see Golub and Van Loan 1989, pp. 344–345). For matrices *A* such that the computed array of normalized eigenvectors *X* is invertible, the condition number of λ_j is

$$\kappa_j = \left\| e_j^T X^{-1} \right\| ,$$

the Euclidean length of the *j*-th row of X^{-1}. Users can choose to compute this matrix using function EIG on page 92. An approximate bound for the accuracy of a computed eigenvalue is then given by

$$\kappa \in \| A \| .$$

To compute an approximate bound for the relative accuracy of an eigenvalue, divide this bound by $| \lambda_j |$.

Reformulating Generalized Eigenvalue Problems

The generalized eigenvalue problem $Ax = \lambda Bx$ is often difficult for users to analyze because it is frequently ill-conditioned. Occasionally, there are changes of variables that can be performed on the given problem to ease this ill-conditioning. Suppose that *B* is singular, but *A* is nonsingular. Define the reciprocal μ

$= \lambda^{-1}$. Then, the roles of A and B are interchanged so that the reformulated problem $Bx = \mu Ax$ is solved. Those generalized eigenvalues $\mu_j = 0$ correspond to eigenvalues $\lambda_j = infinity$. The remaining $\lambda_j = \mu_j^{-1}$. The generalized eigenvectors for λ_j correspond to those for μ_j.

Now, suppose that B is nonsingular. The user can solve the ordinary eigenvalue problem $Cx = \lambda x$, where $C = B^{-1}A$. Matrix C is subject to perturbations due to ill-conditioning and rounding errors when computing $B^{-1}A$. Computing the condition numbers of the eigenvalues for C may, however, be helpful for analyzing the accuracy of results for the generalized problem.

There is another method that users can consider to reduce the generalized problem to an alternate ordinary problem. This technique is based on first computing a matrix decomposition $B = PQ$, where both P and Q are matrices that are "simple" to invert. Then, the given generalized problem is equivalent to the ordinary eigenvalue problem $Fy = \lambda y$. The matrix $F = P^{-1}AQ^{-1}$ and the unnormalized eigenvectors of the generalized problem are given by $x = Q^{-1}y$. An example of this reformulation is used in the case where A and B are real and symmetric, with B positive definite. Function EIGSYMGEN, documented on page 95, uses $P = R^T$ and $Q = R$, where R is an upper-triangular matrix obtained from a Cholesky decomposition, $B = R^TR$. The matrix $F = R^{-T}AR^{-1}$ is symmetric and real. Computation of the eigenvalue-eigenvector expansion for F is based on function EIG.

EIG Function

Computes the eigenexpansion of a real or complex matrix *A*. If the matrix is known to be symmetric or Hermitian, a keyword can be used to trigger more efficient algorithms.

Usage

result = EIG(*a*)

Input Parameters

a — Two-dimensional matrix containing the data.

Returned Value

result — A one-dimensional matrix containing the complex eigenvalues of the matrix.

Input Keywords

Double — If present and nonzero, double precision is used.

Symmetric — If present and nonzero, *a* is assumed to be symmetric in the real case and Hermitian in the complex case. Using keyword *Symmetric* triggers the use of a more appropriate algorithm for symmetric and Hermitian matrices.

Lower_Limit — Used with the keywords *Upper_Limit* and *Symmetric* to force the function EIG to return the eigenvalues and optionally, eigenvectors that lie in the interval with lower limit *Lower_Limit* and upper limit *Upper_Limit.* This keyword can be used only in those cases when *Symmetric* and *Upper_Limit* also are specified.

Default: (*Lower_Limit*, *Upper_Limit*) = (*–infinity*, *+infinity*)

Upper_Limit — Used with the keywords *Lower_Limit* and *Symmetric* to force the function EIG to return the eigenvalues and optionally, eigenvectors that lie in the interval with lower limit *Lower_Limit* and upper limit *Upper_Limit.* This keyword can be used only in those cases when *Symmetric* and *Lower_Limit* also are specified.

Default: (*Lower_Limit*, *Upper_Limit*) = (*–infinity*, *+infinity*)

Output Keywords

Vectors — Specifies the named variable into which the two-dimensional array containing the eigenvectors of the matrix *a* is stored.

Number — Number of eigenvalues and eigenvectors in the range (*Lower_Limit, Upper_Limit*). This keyword is only available if the keyword *Symmetric* also is used. To use this keyword, *Number* and *Symmetric* must be used.

Discussion

If *A* is a real, general matrix, function EIG computes the eigenvalues of *A* by a two-phase process. The matrix is reduced to upper Hessenberg form by elementary orthogonal or Gauss similarity transformations. Then, the eigenvalues are computed using a *QR* or combined *LR-QR* algorithm (Golub and Van Loan 1989, pp. 373–382, and Watkins and Elsner 1990). The combined *LR-QR* algorithm is based on an implementation by Jeff Haag and David Watkins. Eigenvectors are then calculated as required. When eigenvectors are computed, the *QR* algorithm is used to compute the eigenexpansion. When only eigenvalues are required, the combined *LR-QR* algorithm is used.

If *A* is a complex, general matrix, function EIG computes the eigenvalues of *A* by a two-phase process. The matrix is reduced to upper Hessenberg form by elementary Gauss transformations. Then, the eigenvalues are computed using an explicitly shifted *LR* algorithm. Eigenvectors are calculated during the iterations for the eigenvalues (Martin and Wilkinson 1971).

If *A* is a real, symmetric matrix and keyword *Symmetric* is used, function EIG computes the eigenvalues of *A* by a two-phase process. The matrix is reduced to tridiagonal form by elementary orthogonal similarity transformations. Then, the eigenvalues are computed using a rational *QR* or bisection algorithm. Eigenvectors are calculated as required (see Parlett 1980, pp. 169–173).

If *A* is a complex, Hermitian matrix and keyword *Symmetric* is used, function EIG computes the eigenvalues of *A* by a two-phase process. The matrix is reduced to tridiagonal form by elementary orthogonal similarity transformations. Then, the eigenvalues are computed using a rational *QR* or bisection algorithm. Eigenvectors are calculated as required.

If keyword *Symmetric* is used, it is possible to force function EIG to return the eigenvalues and optionally, eigenvectors that lie in a specified interval. The interval is defined using keywords *Lower_Limit* and *Upper_Limit*. The *Number* keyword is provided to return the number of elements of the returned array that

contain valid eigenvalues. The first *Number* elements of the returned array contain the computed eigenvalues, and all remaining elements contain NaN (Not a Number).

Example 1

In this example, the eigenvalues of a real 3-by-3 matrix are computed.

```
RM, a, 3, 3
    ; Define the matrix.
row 0:  8  -1  -5
row 1: -4   4  -2
row 2: 18  -5  -7
eigval = EIG(a)
    ; Call EIG to compute the eigenvalues.
PM, eigval, Title = 'Eigenvalues of A'
    ; Output the results.
Eigenvalues of A
( 2.00000,  4.00001)
( 2.00000, -4.00001)
( 1.00000,  0.00000)
```

Example 2: Computing Eigenvectors

This example is a variation of the first example. Here, the eigenvectors are computed as well as the eigenvalues.

```
RM, a, 3, 3
    ; Define the 3-by-3 matrix.
row 0:  8 -1 -5
row 1: -4  4 -2
row 2: 18 -5 -7
eigval = EIG(a, Vectors = eigvec)
    ; Call EIG using keyword Vectors to specify the named variable into
    ; which the eigenvectors are stored.
PM, eigval, Title = 'Eigenvalues of A'
    ; Output the eigenvalues.


Eigenvalues of A
( 2.00000,  4.00000)
( 2.00000, -4.00000)
```

```
( 1.00001,  0.00000)
```

```
PM, eigvec, Title = 'Eigenvectors of A'
```

; Output the eigenvectors.

```
Eigenvectors of A
( 0.316228, 0.316228)( 0.316228, -0.316228)
( 0.408248, 0.00000)
( 2.08616e-07, 0.632455)( 2.08616e-07, -0.632455)
( 0.816497, 0.00000)
( 0.632456, 0.00000)( 0.632456, 0.00000)
( 0.408247, 0.00000)
```

Example 3: Computing Eigenvalues of a Complex Matrix

```
RM, a, 4, 4, /Complex
```

; Define a complex matrix.

```
row 0: (5, 9) (5,  5) (-6, -6) (-7, -7)
row 1: (3, 3) (6, 10) (-5, -5) (-6, -6)
row 2: (2, 2) (3,  3) (-1,  3) (-5, -5)
row 3: (1, 1) (2,  2) (-3, -3) ( 0,  4)
```

```
eigval = EIG(a)
```

; Call EIG to compute the eigenvalues.

```
PM, eigval, Title = 'Eigenvalues of A'
```

; Output the results.

```
Eigenvalues of A
( 4.00000, 8.00000)
( 3.00000, 7.00000)
( 2.00000, 6.00000)
( 1.00000, 5.00000)
```

Warning Errors

MATH_SLOW_CONVERGENCE_GEN — Iteration for an eigenvalue did not converge after # iterations.

EIGSYMGEN Function

Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$. The matrices *A* and *B* are real and symmetric, and *B* is positive definite.

Usage

result = EIGSYMGEN(*a*, *b*)

Input Parameters

a — Two-dimensional matrix containing the symmetric coefficient matrix *A*.

b — Two-dimensional matrix containing the positive definite symmetric coefficient matrix *B*.

Returned Value

result — One-dimensional array containing the eigenvalues of the symmetric matrix.

Input Keywords

Double — If present and nonzero, double precision is used.

Output Keywords

Vectors — Compute eigenvectors of the problem. A two-dimensional array containing the eigenvectors is returned in the variable name specified by *Vectors*.

Discussion

Function EIGSYMGEN computes the eigenvalues of a symmetric, positive definite eigenvalue problem by a three-phase process (Martin and Wilkinson 1971). Matrix *B* is reduced to factored form using the Cholesky decomposition. These factors are used to form a congruence transformation that yields a symmetric real matrix whose eigenexpansion is obtained. The problem is then transformed back to the original coordinates. Eigenvectors are calculated and transformed as required.

Example 1

In this example, the generalized eigenexpansion of a system $Ax = \lambda Bx$, where A and B are 3-by-3 matrices, is computed.

```
RM, a, 3, 3
    ; Define the matrix A.
row 0: 1.1 1.2 1.4
row 1: 1.2 1.3 1.5
row 2: 1.4 1.5 1.6
RM, b, 3, 3
    ; Define the matrix B.
row 0: 2 1 0
row 1: 1 2 1
row 2: 0 1 2
eigval = EIGSYMGEN(a, b)
    ; Call EIGSYMGEN to compute the eigenexpansion.
PM, eigval, Title = 'Eigenvalues'
    ; Output the results.
Eigenvalues
 1.38644
-0.0583479
-0.00309042
```

Example 2

This example is a variation of the first example. Here, the eigenvectors are computed as well as the eigenvalues.

```
RM, a, 3, 3
    ; Define the matrix A.
row 0: 1.1 1.2 1.4
row 1: 1.2 1.3 1.5
row 2: 1.4 1.5 1.6
RM, b, 3, 3
    ; Define the matrix B.
row 0: 2 1 0
row 1: 1 2 1
row 2: 0 1 2
```

```
eigval = EIGSYMGEN(a, b, Vectors = eigvec)
    ; Call EIGSYMGEN with keyword Vectors to specify the named
    ; variable in which the vectors are stored.
PM, eigval, Title = 'Eigenvalues'
    ; Output the eigenvalues.


Eigenvalues
 1.38644
 -0.0583478
 -0.00309040
PM, eigvec, Title = 'Eigenvectors'
    ; Output the eigenvectors.
Eigenvectors
 0.643094      -0.114730     -0.681688
-0.0223849     -0.687186      0.726597
 0.765460       0.717365     -0.0857800
```

Warning Errors

MATH_SLOW_CONVERGENCE_SYM — Iteration for an eigenvalue failed to converge in 100 iterations before deflating.

Fatal Errors

MATH_SUBMATRIX_NOT_POS_DEFINITE — Leading submatrix of the input matrix is not positive definite.

MATH_MATRIX_B_NOT_POS_DEFINITE — Matrix *B* is not positive definite.

GENEIG Procedure

Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$.

Usage

GENEIG, *a*, *b*, *alpha*, *beta*

Input Parameters

a — Two-dimensional array of size *n*-by-*n* containing the coefficient matrix *A*.

b — Two-dimensional array of size *n*-by-*n* containing the coefficient matrix *B*.

Output Parameters

alpha — One-dimensional array of size *n* containing scalars α_i. If $\beta_i \neq 0$, $\lambda_i = \alpha_i / \beta_i$ *for* $i = 0, \ldots, n - 1$ are the eigenvalues of the system.

beta — One-dimensional array of size *n*.

Input Keywords

Double — If present and nonzero, double precision is used.

Output Keywords

Vectors — Named variable into which a two-dimensional array of size *n*-by-*n* containing eigenvectors of the problem is stored. Each vector is normalized to have Euclidean length equal to one.

Discussion

The function GENEIG uses the QZ algorithm to compute the eigenvalues and eigenvectors of the generalized eigensystem $Ax = \lambda Bx$, where *A* and *B* are matrices of order *n*. The eigenvalues for this problem can be infinite, so α and β are returned instead of λ. If β is nonzero, $\lambda = \alpha/\beta$.

The first step of the QZ algorithm is to simultaneously reduce *A* to upper-Hessenberg form and *B* to upper-triangular form. Then, orthogonal transformations are used to reduce *A* to quasi-upper-triangular form while keeping *B* upper tri-

angular. The generalized eigenvalues and eigenvectors for the reduced problem are then computed.

The function GENEIG is based on the QZ algorithm due to Moler and Stewart (1973), as implemented by the EISPACK routines QZHES, QZIT and QZVAL; see Garbow et al. (1977).

Example 1

In this example, the eigenvalue, λ, of system $Ax = \lambda Bx$ is computed, where

$$A = \begin{bmatrix} 1.0 & 0.5 & 0.0 \\ -10.0 & 2.0 & 0.0 \\ 5.0 & 1.0 & 0.5 \end{bmatrix} \text{ and } B = \begin{bmatrix} 0.5 & 0.0 & 0.0 \\ 3.0 & 3.0 & 0.0 \\ 4.0 & 0.5 & 1.0 \end{bmatrix}$$

```
a   =   TRANSPOSE([[1.0, 0.5, 0.0], $
                   [-10.0, 2.0, 0.0], $
                   [5.0, 1.0, 0.5]])
b   =   TRANSPOSE([[0.5, 0.0, 0.0], $
                   [3.0, 3.0, 0.0], $
                   [4.0, 0.5, 1.0]])
    ; Compute eigenvalues
GENEIG, a, b, alpha, beta
    ; Print eigenvalues
PM, alpha/beta, Title = "Eigenvalues"
Eigenvalues
(      0.833334,        1.99304)
(      0.833333,       -1.99304)
(      0.500000,        0.00000)
```

Example 2

This example finds the eigenvalues and eigenvectors of the same eigensystem given in the last example.

```
a   =   TRANSPOSE([[1.0, 0.5, 0.0], $
                    [-10.0, 2.0, 0.0], $
                    [5.0, 1.0, 0.5]])
```

```
b   =   TRANSPOSE([[0.5,  0.0,  0.0], $
                   [3.0,  3.0,  0.0], $
                   [4.0,  0.5,  1.0]])
    ; Compute eigenvalues
GENEIG, a, b, alpha, beta, Vectors = vectors
    ; Print eigenvalues
PM, alpha/beta, Title = "Eigenvalues"
Eigenvalues
(      0.833332,       1.99304)
(      0.833332,      -1.99304)
(      0.500000,      -0.00000)
    ; Print eigenvectors
PM, vectors, Title = "Eigenvectors"
Eigenvectors
(     -0.197112,       0.149911)(     -0.197112,     -0.149911)
( -1.53306e-08,        0.00000)
(    -0.0688163,     -0.567750)(    -0.0688163,      0.567750)
( -4.75248e-07,        0.00000)
(      0.782047,       0.00000)(      0.782047,       0.00000)
(       1.00000,       0.00000)
```

Example 3

In this example, the eigenvalue, λ, of system $Ax = \lambda Bx$ is solved, where

$$A = \begin{bmatrix} 1 & 0.5+i & 5i \\ -10 & 2+i & 0 \\ 5+i & 1 & 0.5+3i \end{bmatrix} \text{ and } B = \begin{bmatrix} 0.5 & 0 & 0 \\ 3+3i & 3+3i & i \\ 4+2i & 0.5+i & 1+i \end{bmatrix}$$

```
a   =   TRANSPOSE([$
  [COMPLEX(1.0,  0.0), COMPLEX(0.5,  1.0), COMPLEX(0.0,  5.0)], $
  [COMPLEX(-10.0,  0.0), COMPLEX(2.0,  1.0), COMPLEX(0.0,  0.0)], $
  [COMPLEX(5.0,  1.0), COMPLEX(1.0,  0.0), COMPLEX(0.5,  3.0)]])
b   =   TRANSPOSE([$
  [COMPLEX(0.5,  0.0), COMPLEX(0.0,  0.0), COMPLEX(0.0,  0.0)], $
```

```
 [COMPLEX(3.0, 3.0), COMPLEX(3.0, 3.0), COMPLEX(0.0, 1.0)], $
 [COMPLEX(4.0, 2.0), COMPLEX(0.5, 1.0), COMPLEX(1.0, 1.0)]])
   ; Compute eigenvalues
GENEIG, a, b, alpha, beta
   ; Print eigenvalues
PM, alpha/beta, Title  =  "Eigenvalues"
Eigenvalues
(      -8.18016,      -25.3799)
(       2.18006,      0.609113)
(      0.120108,     -0.389223)
```

Example 4

This example finds the eigenvalues and eigenvectors of the same eigensystem given in the last example.

```
a  =  TRANSPOSE([$
 [COMPLEX(1.0, 0.0), COMPLEX(0.5, 1.0), COMPLEX(0.0, 5.0)], $
 [COMPLEX(-10.0, 0.0), COMPLEX(2.0, 1.0), COMPLEX(0.0, 0.0)], $
 [COMPLEX(5.0, 1.0), COMPLEX(1.0, 0.0), COMPLEX(0.5, 3.0)]])
b  =  TRANSPOSE([$
 [COMPLEX(0.5, 0.0), COMPLEX(0.0,0.0), COMPLEX(0.0, 0.0)], $
 [COMPLEX(3.0,3.0), COMPLEX(3.0,3.0), COMPLEX(0.0, 1.0)], $
 [COMPLEX(4.0, 2.0), COMPLEX(0.5, 1.0), COMPLEX(1.0, 1.0)]])
   ; Compute eigenvalues
GENEIG, a, b, alpha, beta, Vectors = vectors
   ; Print eigenvalues
PM, alpha/beta, Title = "Eigenvalues"
Eigenvalues
(      -8.18018,      -25.3799)
(       2.18006,      0.609112)
(      0.120109,     -0.389223)
   ; Print eigenvecters
PM, vectors, Title = "Eigenvectors"
Eigenvectors
(     -0.326709,     -0.124509)(     -0.300678,     -0.244401)
```

```
(     0.0370698,       0.151778)
(      0.176670,    0.00537758)(      0.895923,        0.00000)
(      0.957678,        0.00000)
(      0.920064,        0.00000)(     -0.201900,     0.0801192)
(     -0.221511,     0.0968290)
```

3

CHAPTER

Interpolation and Approximation

Contents of Chapter

Smooth one-dimensional data by error detection..... SMOOTHDATA1D Function

Scattered Data Interpolation

Akima's surface-fitting method ..... SCAT2DINTERP Function

Computes a fit using radial-basis functions ..... RADBF Function

Evaluates a radial-basis fit..... RADBE Function

Linear interpolation of vectors ..... INTERPOL Function

Bilinear interpolation at a set of reference points ..... BILINEAR Function

*For more information on *Standard* Library routines, see "Where to Find PV-WAVE's Libraries" in the *PV-WAVE Programmer's Guide*.

Introduction

Many functions in this chapter produce cubic piecewise polynomial or general spline functions that either interpolate or approximate given data or are support functions for the evaluation and integration of these functions. Three major subdivisions of functions are provided. The cubic spline functions begin with the prefix CS and use the piecewise polynomial representation. The spline functions begin with the prefix BS and use the B-spline representation. The third major subdivision includes functions that operate on the output of both the cubic spline and B-spline functions. Most of the spline functions are based on routines documented by de Boor (1978).

General purpose routines also are provided for general least-squares fit to data and routines to interpolate or approximate scattered data in R^n for $n \geq 1$.

Piecewise Polynomials

A univariate piecewise polynomial function, p, is specified by giving its breakpoint sequence $\xi \in R^n$, the order k (degree $k - 1$) of its polynomial pieces, and the
k x $(n - 1)$ matrix C of its local polynomial coefficients. In terms of this information, the piecewise polynomial (ppoly) function is given by the following equation:

$$p(x) = \sum_{j=1}^{k} c_{ij} \frac{(x - \xi_i)^{j-1}}{(j-1)!} \qquad \text{for} \quad \xi_i \leq x \leq \xi_{i+1}$$

The breakpoint sequence ξ is assumed to be strictly increasing, and the ppoly function is extended to the entire real axis by extrapolation from the first and last intervals. This representation is redundant when the ppoly function is known to be smooth. For example, if p is known to be continuous, then $c_{1,\,i+1}$ can be computed from the c_{ji} as follows:

$$c_{1,\,i+1} = p(\xi_{i+1}) = \sum_{j=1}^{k} c_{ij} \frac{(\xi_{i+1} - \xi_i)^{j-1}}{(j-1)!}$$

For smooth ppoly, the nonredundant representation is used in terms of the "basis" or B-splines, at least when such a function is first to be determined.

Splines and B-splines

B-splines provide a particularly convenient and suitable basis for a given class of smooth ppoly functions. Such a class is specified by giving its breakpoint sequence, its order *k*, and the required smoothness across each of the interior breakpoints. The corresponding B-spline basis is specified by giving its knot sequence $\mathbf{t} \in R^M$. The specification rule is the following: If the class is to have all derivatives up to and including the *j*-th derivative continuous across the interior breakpoint ξ_i, then the number ξ_i should occur $k - j - 1$ times in the knot sequence. Assuming that ξ_1 and ξ_n are the endpoints of the interval of interest, choose the first *k* knots equal to ξ_1 and the last *k* knots equal to ξ_n. This can be done since the B-splines are defined to be right continuous near ξ_1 and left continuous near ξ_n.

When the above construction is completed, a knot sequence **t** of length *M* is generated and there are $m := M - k$ B-splines of order *k* (for example, $B_0, \ldots, B_{m-1}$) that span the ppoly functions on the interval with the indicated smoothness. That is, each ppoly function in this class has a unique representation

$$p = a_0 B_0 + a_1 B_1 + \ldots + a_{m-1} B_{m-1}$$

as a linear combination of B-splines. A B-spline is a particularly compact ppoly function. The function B_i is a nonnegative function that is nonzero only on the interval $[\mathbf{t}_i, \mathbf{t}_{i+k}]$. More precisely, the support of the *i*-th B-spline is $[\mathbf{t}_i, \mathbf{t}_{i+k}]$. No ppoly function in the same class (other than the zero function) has smaller support (i.e., vanishes on more intervals) than a B-spline. This makes B-splines particularly attractive basis functions since the influence of any particular B-spline coefficient extends only over a few intervals. When it is necessary to emphasize the dependence of the B-spline on its parameters, the notation

$$B_{i,k,t}$$

is used to denote the *i*-th B-spline of order *k* for the knot sequence **t**.

Cubic Splines

Cubic splines are smooth (i.e., C^1 or C^2), fourth-order ppoly functions. For historical and other reasons, cubic splines are the most frequently used ppoly functions. Therefore, special functions are provided for their construction and evaluation. These routines use the ppoly representation as described above for general ppoly functions (with $k = 4$).

Two cubic spline interpolation functions, CSINTERP and CSSHAPE, are provided. Function CSINTERP allows the user to specify various endpoint conditions (such as the value of the first or second derivative at the right and left points). This means that the natural cubic spline can be obtained using this function by setting the second derivative to zero at both endpoints. Function CSSHAPE is designed so that the shape of the curve matches the shape of the data. In particular, one option of this function preserves the convexity of the data while the default attempts to minimize oscillations.

It is possible that the cubic spline interpolation functions will produce unsatisfactory results. For example, the interpolant may not have the shape required by the user, or the data may be noisy and require a least-squares fit. The BSINTERP interpolation function is more flexible, as it allows the user to choose the knots and order of the spline interpolant. The user is encouraged to use this routine and exploit the flexibility provided.

Tensor-product Splines

The simplest method of obtaining multivariate interpolation and approximation functions is to take univariate methods and form a multivariate method via tensor products. In the case of two-dimensional spline interpolation, the derivation proceeds as follows: Let $\mathbf{t}_x$ be a knot sequence for splines of order k_x and $\mathbf{t}_y$ be a knot sequence for splines of order k_y. Let $N_x + k_x$ be the length of $\mathbf{t}_x$ and $N_y + k_y$ be the length of $\mathbf{t}_y$. Then, the tensor-product spline has the following form:

$$\sum_{}^{N_y - 1} \sum_{}^{N_x - 1} c_{nm} B_{n, k_x, \mathbf{t}_x}(x) B_{m, k_y, \mathbf{t}_y}(y)$$

Given two sets of points,

$$\{x_i\}_{i=1}^{N_x} \quad and \quad \{y_i\}_{i=1}^{N_y},$$

for which the corresponding univariate interpolation problem can be solved, the tensor-product interpolation problem finds the coefficients c_{nm}, so that the following is true:

$$\sum_{}^{N_y - 1} \sum_{}^{N_x - 1} c_{nm} B_{n, k_x, \mathbf{t}_x}(x_i) B_{m, k_y, \mathbf{t}_y}(y_j) = f_{ij}$$

This problem can be solved efficiently by repeatedly solving univariate interpolation problems as described in de Boor (1978, p. 347). Three-dimensional interpolation can be handled in an analogous manner. This chapter provides functions that compute the two-dimensional, tensor-product spline coefficients given two-dimensional interpolation data (BSINTERP) and functions that compute the two-dimensional, tensor-product spline coefficients for a tensor-product, least-squares problem (BSLSQ). In addition, evaluation, differentiation, and integration routines (SPVALUE and SPINTEG) are provided for the two-dimensional, tensor-product spline functions.

Scattered-data Interpolation and Approximation

PV-WAVE:IMSL Mathematics provides functions to interpolate and approximate scattered data in R^n for $n \geq 1$. Function SCAT2DINTERP interpolates scattered data in the plane and is based on work by Akima (1978), which uses C^1 piecewise quintics on a triangular mesh. Function RADBF can be used to either interpolate or approximate scattered data in R^n for $n \geq 1$. The RADBF function computes approximations based on radial-basis functions. The fit computed by RADBF can be evaluated using function RADBE.

Least Squares

PV-WAVE:IMSL Mathematics includes functions for smoothing noisy data. Function FCNLSQ computes regressions with user-supplied functions. Function BSLSQ computes a one- or two-dimensional, least-squares fit using splines with fixed knots or variable knots. This function produces cubic-spline, least-squares fit by default. Keywords allow the user to choose the order and the knot sequence.

PV-WAVE:IMSL Statistics contains many functions that provide for polynomial regression and general linear regression.

Smoothing by Cubic Splines

One “smoothing spline” function is provided. The default action of CSSMOOTH estimates a smoothing parameter by cross-validation, then returns the cubic spline that smooths the data. If the user chooses to supply a smoothing parameter, this function returns the appropriate cubic spline.

Structures for Splines and Piecewise Polynomials

This section is optional and is intended for users interested in more details concerning the structures for splines and piecewise polynomials.

A spline can be viewed as a mapping with domain R^d and target R^r, where d and r are positive integers. For this version of PV-WAVE:IMSL Mathematics, only $r = 1$ is supported. Thus, if s is a spline, then the following is true for some d and r:

$$s: R^d \rightarrow R^r$$

This implies that such a spline s must have d knot sequences and orders (one for each domain dimension). Thus, associated with s, knots and orders are as follows:

$$\mathbf{t}^0, ..., \mathbf{t}^{d-1}$$

$$k_0, ..., k_{d-1}$$

The precise form of the spline follows:

$$s(x) = (s_0(x), ..., s_{r-1}(x)) \qquad x = (x_1, ..., x_d) \in R^d$$

where

$$s_i(x) := \sum_{j_{d-1}=0}^{n_{d-1}-1} \cdots \sum_{j_0=0}^{n_0-1} c^i_{j_0, \ldots, j_{d-1}} B_{j_0, k_0, \mathbf{t}^0} \cdots B_{j_{d-1}, k_{d-1}, \mathbf{t}^{d-1}}$$

Note that n_i is the number of knots in $\mathbf{t}^i$ minus the order k_i.

All the information for a spline is stored in a structure. Since, in general, the components of this structure are of varying lengths, an anonymous structure is defined for each spline. An example of the information returned by the INFO command with keyword *Structures* set and an argument containing a spline structure follows:

```
x = FINDGEN(10)
y = RANDOM(10)
spline = BSINTERP(x, y)
INFO, spline, /Structure
** Structure $1, 7 tags, 116 length:
DOMAIN_DIM      LONG        1
TARGET_DIM      LONG        1
```

```
ORDER              LONG        4
NUM_COEF           LONG       10
NUM_KNOTS          LONG       14
KNOTS              FLOAT      Array(14)
COEF               FLOAT      Array(10)
```

For ppoly functions, a ppoly is viewed as a mapping with domain R^d and target R^r, where d and r are positive integers. Thus, if p is a ppoly, then the following is true for some d and r:

$$p\colon R^d \rightarrow R^r$$

For this version of PV-WAVE:IMSL Mathematics, only $r = 1$ is supported. This implies that such a ppoly p must have d breakpoint sequences and orders (one for each domain dimension). Thus, associated with p, breakpoints and orders are as follows:

$$\xi^1, \ldots, \xi^d$$

$$k_1, \ldots, k_d$$

The precise form of the ppoly follows:

$$p(x) = (p_0(x), \ldots, p_r(x)) \qquad x = (x_1, \ldots, x_d) \in R^d$$

where

$$p_i(x) := \sum_{l_d = 0}^{k_d - 1} \cdots \sum_{l_1 = 0}^{k_1 - 1} c^i_{L^1, \ldots, L^d, l_1, \ldots, l_d} \frac{(x_1 - \xi^1_{L^1})^{l_1}}{l_1!} \cdots \frac{(x_d - \xi^d_{L^d})^{l_d}}{l_d!}$$

with

$$L^j := \max\{1, \min\{M^j, n_j - 1\}\}$$

where M^j is chosen so that

$$\xi^j_{M^j} \le x_j < \xi^j_{M^j + 1} \qquad j = 1, \ldots, d$$

(with $\xi^j_0 = -\infty$ and $\xi^j_{n_j + 1} = \infty$).

Note that n_j is the number of breakpoints in ξ^j.

All the information for a spline is stored in a structure. Since, in general, the components of this structure are of varying lengths, an anonymous structure is defined for each spline. An example of the information returned by the INFO command with keyword *Structures* set and an argument containing a spline structure is as follows:

```
x = FINDGEN(10)
y = RANDOM(10)
ppoly = CSINTERP(x, y)
INFO, ppoly, /Structure
** Structure $2, 7 tags, 204 length:
   DOMAIN_DIM       LONG        1
   TARGET_DIM       LONG       1
   ORDER             LONG        4
   NUM_COEF          LONG        36
   NUM_BREAKPOINTS LONG        10
   BREAKPOINTS      FLOAT      Array(10)
   COEF              FLOAT      Array(36)
```

CSINTERP Function

Computes a cubic spline interpolant, specifying various endpoint conditions. The default interpolant satisfies the not-a-knot condition.

Usage

result = CSINTERP(*xdata*, *fdata*)

Input Parameters

xdata — One-dimensional array containing the abscissas of the interpolation problem.

fdata — One-dimensional array containing the ordinates for the interpolation problem.

Returned Value

result — A structure that represents the cubic spline interpolant.

Input Keywords

Double — If present and nonzero, double precision is used.

ILeft — Sets the value for the first or second derivative of the interpolant at the left endpoint. Keyword *ILeft* is used to specify which derivative is set: *ILeft* = 1 for the first derivative and *ILeft* = 2 for the second derivative. The only valid values for *ILeft* are 1 or 2. If *ILeft* is specified, then *Left* also must be used.

Left — Sets the value for the first or second derivative of the interpolant at the left endpoint. If *ILeft* = *i*, then the interpolant *s* satisfies $s^{(i)}(x_L) = Left$. Here, x_L is the leftmost abscissa.

IRight — Sets the value for the first or second derivative of the interpolant at the right endpoint. Keyword *IRight* is used to specify which derivative is set: *IRight* = 1 for the first derivative and *IRight* = 2 for the second derivative. The only valid values for *IRight* are 1 or 2. If *IRight* is specified, then *Right* also must be used.

Right — Sets the value for the first or second derivative of the interpolant at the right endpoint. If *IRight* = *i*, then the interpolant *s* satisfies $s^{(i)}(x_R) = Right$. Here, x_R is the rightmost abscissa.

Periodic — If present and nonzero, computes the C^2 periodic interpolant to the data. The following is satisfied:

$$s^{(i)}(x_L) = s^{(i)}(x_R) \qquad i = 0, 1, 2$$

where *s*, x_L, and x_R are defined above.

Discussion

Function CSINTERP computes a C^2 cubic spline interpolant to a set of data points (x_i, f_i) for the following:

$$i = 0, \ldots, (\text{N\_ELEMENTS}(xdata) - 1) = (n - 1)$$

The breakpoints of the spline are the abscissas. For all univariate interpolation functions, the abscissas need not be sorted. Endpoint conditions are to be selected by the user. The user can specify not-a-knot, or first or second derivatives at each endpoint or C^2 periodicity can be requested (see de Boor 1978, Chapter 4). If no defaults are selected, then the not-a-knot spline interpolant is computed. If the *Periodic* keyword is selected, then all other keywords are ignored and a C^2 is computed. In this case, if the *fdata* values at the left and right endpoints are not the same, a warning message is issued and the right value is set equal to the left. If *Left* and *ILeft* or *Right* and *IRight* are selected,

the user has the ability to select the values of the first or second derivative at either endpoint. The default case (when the keyword is not used) is the not-a-knot condition on that endpoint. Thus, when no keywords are chosen, this function produces the not-a-knot interpolant.

If the data (including the endpoint conditions) arise from the values of a smooth (for example, C^4) function f, i.e., $f_i = f(x_i)$, then the error behaves in a predictable fashion. Let ξ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\|f - s\|_{[\xi_0, \xi_n]} \leq C \|f^{(4)}\|_{[\xi_0, \xi_n]} |\xi|^4$$

where the following is true:

$$|\xi| := \max_{i = 0, \ldots, n-1} |\xi_{i+1} - \xi_i|$$

Example 1

In this example, a cubic spline interpolant to function values is computed and plotted along with the original data. Since the default settings are used, the interpolant is determined by the not-a-knot condition (see de Boor 1978).

```
x = FINDGEN(11)/10
    ; Generate the abscissas.
f = SIN(15 * x)
    ; Generate the function values.
pp = CSINTERP(x, f)
    ; Compute the spline interpolant.
ppval = SPVALUE(FINDGEN(100)/99, pp)
PLOT, FINDGEN(100)/99, ppval
    ; Plot the results.
OPLOT, x, f, Psym = 6
```

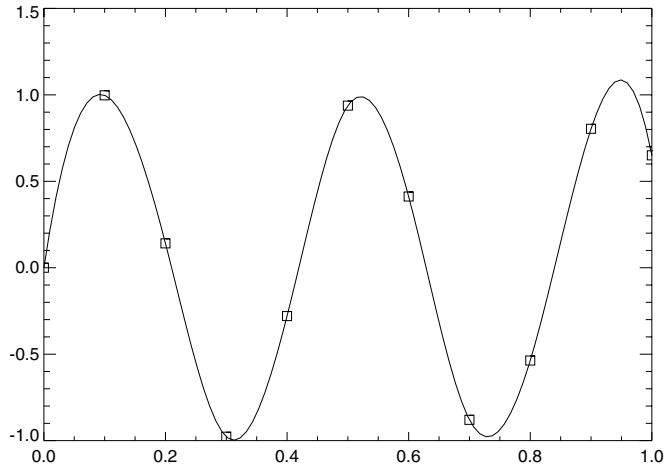


Figure 3-1 Cubic spline interpolant.

Example 2

In this example, a cubic spline interpolant to function values is computed. The value of the derivative at the left endpoint and the value of the second derivative at the right endpoint are specified. The resulting spline and original data are then plotted.

```
x = FINDGEN(11)/10
y = SIN(15 * x)
pp = CSINTERP(x, y, ILeft = 1, Left = 0, $
   IRight = 2, Right = -225 * SIN(15))
ppval = SPVALUE(FINDGEN(100)/99, pp)
PLOT, FINDGEN(100)/99, ppval
OPLOT, x, y, Psym = 6
```

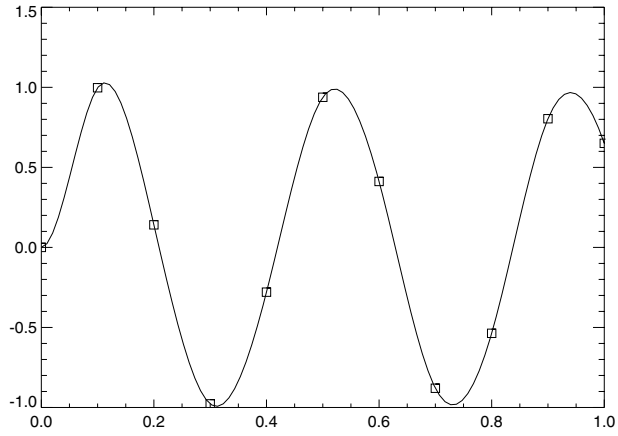


Figure 3-2 Cubic spline interpolant with endpoint conditions specified.

Warning Errors

MATH_NOT_PERIODIC — Data are not periodic. The rightmost *fdata* value is set to the leftmost *fdata* value.

Fatal Errors

MATH_DUPLICATE_XDATA_VALUES — The *xdata* values must be distinct.

CSSHAPE Function

Computes a shape-preserving cubic spline.

Usage

result = CSSHAPE(*xdata*, *fdata*)

Input Parameters

xdata — One-dimensional array containing the abscissas of the interpolation problem.

fdata — One-dimensional array containing the ordinates for the interpolation problem.

Returned Value

result — A structure that represents the cubic spline interpolant.

Input Keywords

Double — If present and nonzero, double precision is used.

Concave — If present and nonzero, CSSHAPE produces a cubic interpolant that preserves the concavity of the data.

Itmax — Allows the user to set the maximum number of iterations of Newton's Method. To use *Itmax*, keyword *Concave* must also be set.

Default: *Itmax* = 25

Discussion

Function CSSHAPE computes a C^1cubic spline interpolant to a set of data points (x_i, f_i) for the following:

$i = 0, \ldots, (\text{N\_ELEMENTS}(xdata) - 1) = (n - 1)$

The breakpoints of the spline are the abscissas. This computation is based on a method by Akima (1970) to combat wiggles in the interpolant. Endpoint conditions are automatically determined by the program (see Akima 1970, de Boor 1978).

If the *Concave* keyword is set, then this function computes a cubic spline interpolant to the data. For ease of explanation, $x_i < x_{i+1}$ is assumed, although it is not necessary for the user to sort these data values. If the data are strictly convex, then the computed spline is convex, C^2, and minimizes the expression

$$\int_{x_1}^{x_n} (g'')^2$$

over all convex C^1functions that interpolate the data. In the general case, when the data have both convex and concave regions, the convexity of the spline is consistent with the data, and the above integral is minimized under the appropriate constraints. For more information on this interpolation scheme, refer to Micchelli et al. (1985) and Irvine et al. (1986).

One important feature of the splines produced by this function is that it is not possible, *a priori*, to predict the number of breakpoints of the resulting interpolant. In most cases, there will be breakpoints at places other than data locations. This function should be used when it is important to preserve the convex and concave regions implied by the data.

Both methods are nonlinear, and although the interpolant is a piecewise cubic, cubic polynomials are not reproduced. (However, linear polynomials are reproduced.) This explains the theoretical error estimate below.

If the data points arise from the values of a smooth (for example, C^4) function f, i.e., $f_i = f(x_i)$, then the error behaves in a predictable fashion. Let ξ be the breakpoint vector for either of the above spline interpolants. Then, the maximum absolute error satisfies

$$\|f - s\|_{[\xi_0, \xi_n]} \leq C \|f^{(2)}\|_{[\xi_0, \xi_n]} |\xi|^2$$

where

$$|\xi| := \max_{i = 0, \ldots, n-1} |\xi_{i+1} - \xi_i|$$

and ξ_m is the last breakpoint.

The returned value for this function is a structure. This structure contains all the information to determine the spline (stored as a piecewise polynomial) that is computed by this function. For example, the following code sequence evaluates this spline at *x* and returns the value in *y*:

```
y = SPVALUE(x, spline)
```

Example 1

In this example, a cubic spline interpolant to function values is computed. Evaluations of the computed spline are plotted along with the original data values.

```
x = FINDGEN(10)/9
    ; Define the abscissas.
f = FLTARR(10)
f(0:4) = 0.25
f(5:9) = 0.75
    ; Define the function values.
pp = CSSHAPE(x, f)
    ; Compute the interpolant.
ppval = SPVALUE(FINDGEN(100)/99, pp)
    ; Evaluate the interpolant at 100 values in [0,1].
PLOT, FINDGEN(100)/99, ppval
    ; Plot the results.
OPLOT, x, f, Psym = 6
```

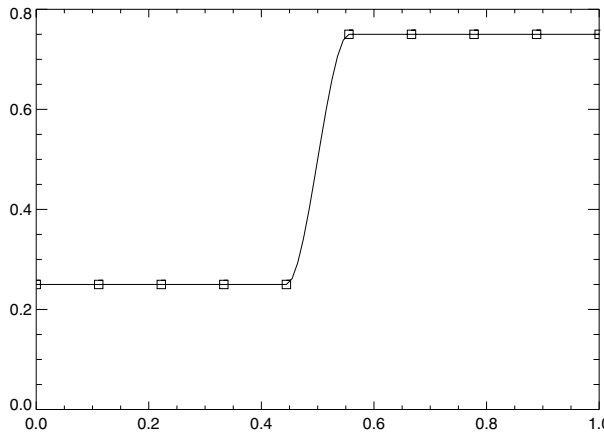


Figure 3-3 Shape-preserving cubic spline.

Example 2

This example compares interpolants computed by CSINTERP (page 111) and CSSHAPE with keyword *Concave*.

```
x = [0, .1, .2, .3, .4, .5, .6, .8, 1]
```

```
y = [0, .9, .95, .9, .1, .05, .05, .2, 1]
    ; Define the data set.
pp1 = CSINTERP(x, y)
    ; Compute the interpolant from CSINTERP.
pp2 = CSSHAPE(x, y, /Concave)
    ; Compute the interpolant from CSSHAPE with keyword Concave.
x2 = FINDGEN(100)/99
PLOT, x2, SPVALUE(x2, pp1), Linestyle = 2
    ; Plot the results.
OPLOT, x2, SPVALUE(x2, pp2)
OPLOT, x, y, Psym = 6
XYOUTS, .4, .9, 'CSINTERP', Charsize = 1.2


OPLOT, [.73, .85], [.925, .925], $
   Linestyle = 2
XYOUTS, .4, .8, 'CSSHAPE !cwith CONCAVE', $
   Charsize = 1.2
OPLOT, [.73, .85], [.8, .8]
XYOUTS, .4, .6, 'Original data', $
   Charsize = 1.2
OPLOT, [.73], [.622], Psym = 6
```

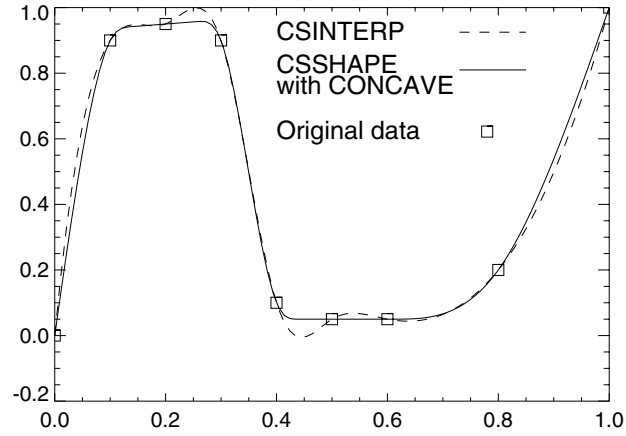


Figure 3-4 Comparison between cubic spline and concavity-preserving cubic spline.

Warning Errors

MATH_MAX_ITERATIONS_REACHED — Maximum number of iterations has been reached. The best approximation is returned.

Fatal Errors

MATH_DUPLICATE_XDATA_VALUES — The *xdata* values must be distinct.

BSINTERP Function

Computes a one- or two-dimensional spline interpolant.

Usage

result = BSINTERP(*xdata*, *fdata*)

result = BSINTERP(*xdata*, *ydata*, *fdata*)

Input Parameters

If a one-dimensional spline is desired, then the arguments *xdata* and *fdata* are required. If a two-dimensional, tensor-product spline is desired, then *xdata*, *ydata*, and *fdata* are required.

xdata — Array containing the abscissas in the *x*-direction of the interpolation problem.

ydata — Array containing the abscissas in the *y*-direction of the interpolation problem.

fdata — Array containing the ordinates of the interpolation problem. If a one-dimensional spline is being computed, then *fdata* (*i*) is the data value at *xdata* (*i*). If a two-dimensional spline is being computed, then *fdata* is a two-dimensional array, where *fdata* (*i, j*) is the data value at (*xdata* (*i*), *ydata* (*i*)).

Returned Value

result — A structure containing information that defines the one- or two-dimensional spline.

Keywords

Double — If present and nonzero, double precision is used.

XOrder — Specifies the order of the spline in the *x*-direction.

Default: *XOrder* = 4, i.e., cubic splines

YOrder — Specifies the order of the spline in the *y*-direction. If a one-dimensional spline is being computed, then *YOrder* has no effect on the computations.

Default: *YOrder* = 4, i.e., cubic splines

XKnots — Specifies the array of knots in the *x*-direction to be used when computing the definition of the spline.

Default: knots are selected by function BSKNOTS using its defaults

YKnots — Specifies the array of knots in the *y*-direction to be used when computing the definition of the spline.

Default: knots are selected by function BSKNOTS using its defaults

Discussion

Function BSINTERP is designed to compute either a one-dimensional spline interpolant or two-dimensional, tensor-product spline interpolant to input data. The decision of whether to compute the one- or two-dimensional spline is based on the number of arguments passed to the function. Keywords are provided to allow the user to specify the order of the spline and the knots used for the spline. When computing a one-dimensional spline, the available keywords are *XOrder* and *XKnots*. When computing a two-dimensional spline, the user can specify the order and knots in *x*-direction and/or *y*-direction using keywords *XOrder, XKnots, YOrder,* and *YKnots.*

Separate discussions on one- and two-dimensional splines follow.

One-dimensional B-splines

Given the data points *x* = *xdata*, *f* = *fdata*, and the number of elements (*n*) in *xdata* and *fdata*, the default action of BSINTERP computes a cubic (*k* = 4) spline interpolant *s* to the data using the default knot sequence generated by function BSKNOTS (page 128).

Optional argument *XOrder* allows the user to choose the order, *k*, of the spline interpolant; optional argument *XKnots* allows user specification of knots.

Function BSINTERP is based on the routine SPLINT by de Boor (1978, p. 204).

First, BSINTERP sorts the *xdata* vector and stores the result in *x*. The elements of the *fdata* vector are permuted appropriately and stored in *f*, yielding the equivalent data (x_i, f_i) for $i = 0$ to $n - 1$.

The following preliminary checks are performed on the data:

$$x_i < x_{i+1} \qquad i = 0, \ldots, n-2$$

$$\mathbf{t}_i < \mathbf{t}_{i+k} \qquad i = 0, \ldots, n-1$$

$$\mathbf{t}_t \leq \mathbf{t}_{i+1} \qquad i = 0, \ldots, n+k-2$$

The first test checks to see that the abscissas are distinct. The second and third inequalities verify that a valid knot sequence has been specified.

In order for the interpolation matrix to be nonsingular, $\mathbf{t}_{k-1} \leq x_i \leq \mathbf{t}_n$ is also checked for $i = 0$ to $n - 1$. This first inequality in the last check is necessary since the method used to generate the entries of the interpolation matrix requires that the k possibly nonzero B-splines at x_i

$$B_{j-k+1}, \ldots, B_j \qquad \text{where } j \text{ satisfies } \mathbf{t}_j \leq x_i < \mathbf{t}_{j+1}$$

be well-defined (that is, $j - k + 1 \geq 0$).

General conditions are not known for the exact behavior of the error in spline interpolation; however, if **t** and *x* are selected properly and the data points arise from the values of a smooth (for example, C^k) function *f*, i.e., $f_i = f(x_i)$, then the error behaves in a predictable fashion. The maximum absolute error satisfies

$$\|f - s\|_{[\mathbf{t}_{k-1}, \mathbf{t}_n]} \leq C \left\|f^{(k)}\right\|_{[\mathbf{t}_{k-1}, \mathbf{t}_n]} |\mathbf{t}|^k$$

where the following is true:

$$|\mathbf{t}| := \max_{i = k-1, \ldots, n-1} |\mathbf{t}_{i+1} - \mathbf{t}_i|$$

For more information on this problem, see de Boor (1978, Chapter 13) and the references therein. This function can be used in place of function CSINTERP (page 111).

The returned value for this function is a structure. This structure contains all the information to determine the spline (stored as a linear combination of

B-splines) that is computed by this function. For example, the following code sequence evaluates this spline at x and returns the value in y:

```
y = SPVALUE(x, spline)
```

Two-dimensional, Tensor-product B-splines

If arguments *xdata*, *ydata*, and *fdata* are all included in the call to function BSINTERP, the function computes a two-dimensional, tensor-product spline interpolant. The tensor-product spline interpolant to data $\{(x_i, y_j, f_{ij})\}$, where $0 \le i \le n_x - 1$ and $0 \le j \le n_y - 1$, has the form

$$\sum_{} \sum_{} c_{nm} B_{n, k_x, \mathbf{t}_x}(x) B_{m, k_y, \mathbf{t}_y}(y)$$

where k_x and k_y are the orders of the splines. These numbers are defaulted to 4 but can be set to any positive integer using keywords *XOrder* and *YOrder*. Likewise, $\mathbf{t}_x$ and $\mathbf{t}_y$ are the corresponding knot sequences (*XKnots* and *YKnots*). These values are defaulted to the knots returned by function BSKNOTS. The algorithm requires that the following is true:

$$\mathbf{t}_x (k_x - 1) \le x_i \le \mathbf{t}_x (n_x) \qquad 0 \le i \le n_x - 1$$

$$\mathbf{t}_y (k_y - 1) \le y_j \le \mathbf{t}_y (n_y) \qquad 0 \le j \le n_y - 1$$

Tensor-product spline interpolants in two dimensions can be computed quite efficiently by solving (repeatedly) two univariate interpolation problems. The computation is motivated by the following observations:

$$\sum_{} \sum_{} c_{nm} B_{n, k_x, \mathbf{t}_x}(x_i) B_{m, k_y, \mathbf{t}_y}(y_j) = f_{ij}$$

Setting

$$h_{mi} = \sum_{n=0}^{n_x - 1} c_{nm} B_{n, k_x, \mathbf{t}_x}(x_i),$$

note that for each fixed i from 0 to $n_x - 1$, there are n_y linear equations in the same number of unknowns as can be seen below.

$$\sum_{} h_{mi} B_{m, k_y, \mathbf{t}_y}(y_i) = f_{ij}$$

The same matrix appears in the equation above.

$$[B_{m,k_y,\mathbf{t}_y}(y_j)] \qquad 1 \le m, j \le n_y - 1$$

Thus, this matrix is factored only once, then the factorization to solve the n_x right-hand sides is applied. Once this is done and h_{mi} is computed, then the coefficients c_{nm} are solved using the relation

$$\sum_{n_x - 1} c_{nm} B_{n,k_x,\mathbf{t}_x}(x_i) = h_{mi}$$

for m from 0 to $n_y - 1$, which involves one factorization and n_y solutions to the different right-hand sides. This ability of function BSINTERP is based on the SPLI2D routine by de Boor (1978, p. 347).

The returned value is a structure containing all the information to determine the spline (stored in B-spline format) that is computed by this function. For example, the following code sequence evaluates this spline at (*x*, *y*) and returns the value in *z*:

```
z = SPVALUE(x, y, spline)
```

Example 1

In this example, a one-dimensional B-spline interpolant to function values is computed. Evaluations of the computed spline are then plotted along with the original data values. Since the default settings are being used, the interpolant is determined by the not-a-knot condition (see de Boor 1978).

```
x = FINDGEN(11)/10
    ; Define data values.
f = SIN(15 * x)
bs = BSINTERP(x, f)
    ; Compute interpolant.
bsval = SPVALUE(FINDGEN(100)/99, bs)
PLOT, FINDGEN(100)/99, bsval
    ; Output results.
OPLOT, x, f, Psym = 6
```

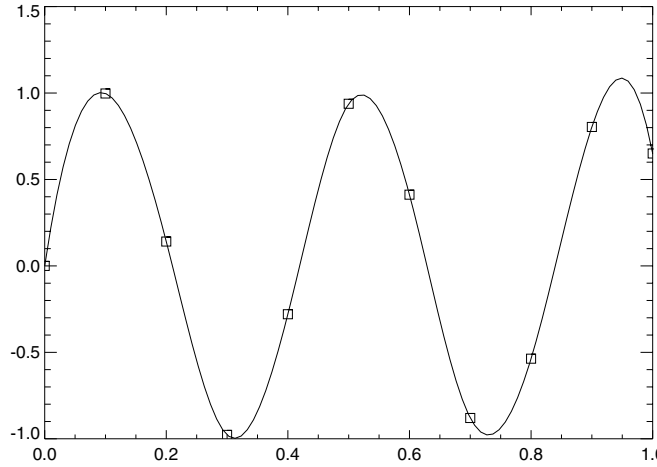


Figure 3-5 B-spline interpolant.

Example 2

In this example, a two-dimensional, tensor-product B-spline interpolant to gridded data is computed.

```
x = FINDGEN(5)/4
```
; Define the abscissas in the *x*-direction.

```
y = FINDGEN(5)/4
```
; Define the abscissas in the *y*-direction.

```
f = FLTARR(5, 5)
```
; Define the sample function values.

```
FOR i = 0, 4 DO $
   f(i, *) = SIN(2 * x(i)) - COS(5 * y)

bs = BSINTERP(x, y, f)
```
; Compute the spline interpolant.

```
bsval = SPVALUE(FINDGEN(20)/19, $
   FINDGEN(20)/19, bs)
```
; Use SPVALUE to evaluate the computed spline.

```
!P.Charsize = 1.5
!P.Multi = [0, 1, 2]

WINDOW, XSize = 400, YSize = 800
```
; Plot the original and computed surfaces in a tall window.

```
SURFACE, f, x, y
```

```
SURFACE, bsval, FINDGEN(20)/19, $
   FINDGEN(20)/19
```

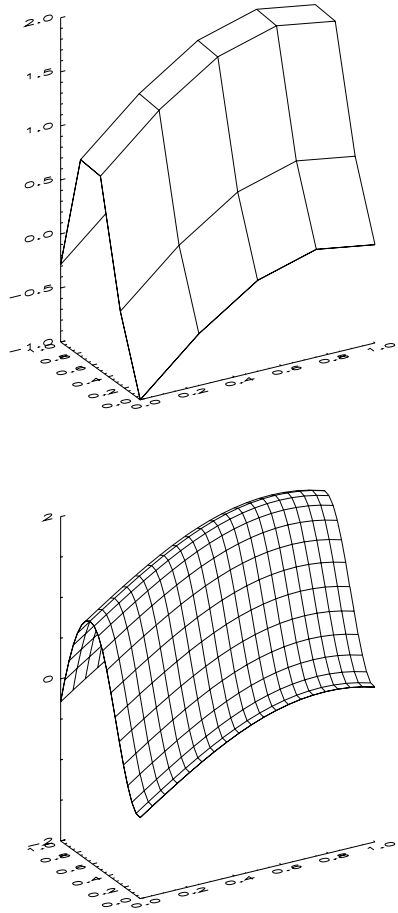


Figure 3-6 Two-dimensional B-spline interpolant to gridded data.

Warning Errors

MATH_ILL_COND_INTERP_PROB — Interpolation matrix is ill-conditioned. Solution might not be accurate.

Fatal Errors

MATH_DUPLICATE_XDATA_VALUES — The *xdata* values must be distinct.

MATH_YDATA_NOT_INCREASING — The *ydata* values must be strictly increasing.

MATH_KNOT_MULTIPLICITY — Multiplicity of the knots cannot exceed the order of the spline.

MATH_KNOT_NOT_INCREASING — Knots must be nondecreasing.

MATH_KNOT_XDATA_INTERLACING — The *i*-th smallest element of *xdata* (x_i) must satisfy $\mathbf{t}_i \le x_i < \mathbf{t}_{i + Order}$, where **t** is the knot sequence.

MATH_XDATA_TOO_LARGE — Array *xdata* must satisfy $xdata_i \le \mathbf{t}_{ndata}$, for $i = 1, \ldots, ndata$.

MATH_XDATA_TOO_SMALL — Array *xdata* must satisfy $xdata_i \ge \mathbf{t}_{Order - 1}$, for $i = 1, \ldots, ndata$.

MATH_KNOT_DATA_INTERLACING — The *i*-th smallest element of the data arrays *xdata* and *ydata* must satisfy $\mathbf{t}_i \le data_{i + Order}$, where **t** is the knot sequence.

MATH_DATA_TOO_LARGE — Data arrays *xdata* and *ydata* must satisfy $data_i \le \mathbf{t}_{num\_data}$, for $i = 1, \ldots, num\_data$.

MATH_DATA_TOO_SMALL — Data arrays *xdata* and *ydata* must satisfy $data_i \ge \mathbf{t}_{Order - 1}$, for $i = 1, \ldots, num\_data$.

BSKNOTS Function

Computes the knots for a spline interpolant.

Usage

result = BSKNOTS(*xdata*)

Input Parameters

xdata — One-dimensional array containing the abscissas of the interpolation problem.

Returned Value

result — A one-dimensional array containing the computed knots.

Input Keywords

Double — If present and nonzero, double precision is used.

Order — Order of the spline subspace for which the knots are desired.

Default: *Order* = 4, i.e., cubic splines

Optimum — If present and nonzero, knots that satisfy an optimal criterion are computed. See *Discussion* for more information.

Itmax — Integer value used to set the maximum number of iterations of Newton's method. To use this keyword, keyword *Optimum* must also be set.

Default: *Itmax* = 10

Discussion

Given the data points x = *xdata*, the order of the spline k = *Order*, and the number n = N_ELEMENTS *(xdata)* of elements in *xdata*, the default action of BSKNOTS returns a knot sequence that is appropriate for interpolation of data on x by splines of order k (the default order is $k = 4$). The knot sequence is contained in its $n + k$ elements. If k is even and it is assumed that the entries in the input vector x are increasing, then the resulting knot sequence **t** is returned as follows:

$$\mathbf{t}_i = x_0 \qquad \text{for } i = 0, \ldots, k-1$$

$$\mathbf{t}_i = x_{i-k/2-1} \quad \text{for } i = k, \ldots, n-1 \quad (1)$$

$$\mathbf{t}_i = x_{n-1} \qquad \text{for } i = n, \ldots, n+k-1$$

There is some discussion concerning this selection of knots in de Boor (1978, p. 211). If *k* is odd, then **t** is returned as follows:

$$\mathbf{t}_i = x_0 \qquad \text{for } i = 0, \ldots, k-1$$

$$\mathbf{t}_i = \frac{1}{2}\left(x_{i-\frac{k-1}{2}-1} + x_{i-1-\frac{k-2}{2}} \right) \qquad \text{for } i = k, \ldots, n-1$$

$$\mathbf{t}_i = x_{n-1} \qquad \text{for } i = n, \ldots, n+k-1$$

It is not necessary to sort the values in *xdata.*

If keyword *Optimum* is set, then the knot sequence returned minimizes the constant *c* in the error estimate

$$\| f - s \| \leq c \| f^{(k)} \|$$

where *f* is any function in C^k and *s* is the spline interpolant to *f* at the abscissa *x* with knot sequence **t**.

The algorithm is based on a routine described by de Boor (1978, p. 204), which in turn is based on a theorem of Micchelli et al. (1976).

Example 1

In this example, knots for a cubic spline are generated and printed. Notice that the knots are stacked at the endpoints; also, the second and next-to-last data points are not knots.

```
x = FINDGEN(6)
knots = BSKNOTS(x)
PM, knots, Format = '(f5.2)'
 0.00
 0.00
 0.00
 0.00
 2.00
 3.00
```

```
 5.00
 5.00
 5.00
 5.00
```

Example 2

This example compares the default knots with the knots returned using keyword *Optimize*. The order also is changed from the default value of 4 to 3.

```
x = FINDGEN(11)/10
    ; Define the abscissa values.
f = FLTARR(11)
    ; Define the function values.
f(0:3) = .25
f(4:7) = .5
f(8:10) = .25
sp1 = BSINTERP(x, f)
    ; Compute the default spline.
knots2 = BSKNOTS(x, /Optimum, Order = 3)
    ; Compute the optimum knots of order 3.
sp2 = BSINTERP(x, f, XKnots = knots2, XOrder = 3)
       ; Compute the spline of order 3, with the optimum knots.
x2 = FINDGEN(100)/99
    ; Evaluate the two splines for plotting.
sp1eval = SPVALUE(x2, sp1)
sp2eval = SPVALUE(x2, sp2)
PLOT, x2, sp1eval, Linestyle = 2
    ; Plot the results.
OPLOT, x2, sp2eval
OPLOT, x, f, Psym = 6
XYOUTS, .25, .18, 'With optimum knots:', $
   Charsize = 1.5
OPLOT, [.65, .75], [.188, .188]
XYOUTS, .25, .135, 'With default knots:', Charsize = 1.5
OPLOT, [.65, .75], [.143, .143], $
   Linestyle = 2
XYOUTS, .3, .09, 'Original data', $
   Charsize = 1.5
```

```
OPLOT, [.7], [.098], Psym = 6
```

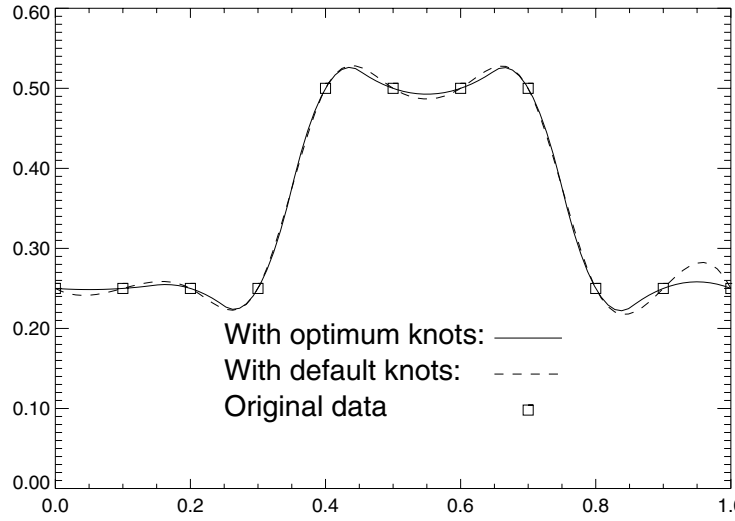


Figure 3-7 Example of optimum knot placement.

Warning Errors

MATH_NO_CONV_NEWTON — Newton's method iteration did not converge.

Fatal Errors

MATH_DUPLICATE_XDATA_VALUES — The *xdata* values must be distinct.

MATH_ILL_COND_LIN_SYS — Interpolation matrix is singular. The *xdata* values may be too close together.

SPVALUE Function

Computes values of a spline or values of one of its derivatives.

Usage

result = SPVALUE(*x*, *spline*)

result = SPVALUE(*x*, *y*, *spline*)

Input Parameters

If evaluation of a one-dimensional spline is desired, then arguments *x* and *spline* are required. If evaluation of a two-dimensional spline is desired, then *x*, *y*, and *spline* are required.

x — Scalar value or an array of values at which the spline is to be evaluated in the *x*-direction. If *x* is an array, then *x* must be strictly increasing, i.e., $x(i) < x(i + 1)$ for $i = 0$, (N_ELEMENTS $(x) - 2$).

y — Scalar value or an array of values at which the spline is to be evaluated in the *y*-direction. This argument should only be used if *spline* is a two-dimensional, tensor-product spline. If *y* is an array, then *x* must be strictly increasing, i.e., $y(i) < y(i + 1)$ for $i = 0$, (N_ELEMENTS $(y) - 2$).

spline — Structure that represents the spline.

Returned Value

result — The values of a spline or one of its derivatives.

Input Keywords

XDeriv — Let *XDeriv* = *p*, and let *s* be the spline that is represented by *spline*.

If *s* is a one-dimensional spline, this keyword produces the *p*-th derivative of *s* at *x*, $s^{(p)}(x)$. If *s* is a two-dimensional spline, this keyword specifies the order of the partial derivative in the *x*-direction. Let *q* = *YDeriv*, which has a default value of 0. Then, SPVALUE produces the (*p*, *q*)-th derivative of *s* at (*x*, *y*), $s^{(p, q)}(x, y)$.

Default: *XDeriv* = 0

YDeriv — If *s* = *spline* is a two-dimensional spline, this keyword specifies the order of the partial derivative in the *y*-direction. Let *p* = *XDeriv*, which has a default value of zero, and *q* = *YDeriv*. Then, SPVALUE produces the (*p*, *q*)-th derivative of *s* at (*x*, *y*), $s^{(p, q)}(x, y)$. If *spline* is a one-dimensional spline, this keyword has no effect on computations.

Default: *YDeriv* = 0

Discussion

Function SPVALUE can be used to evaluate splines of the following type:

- Piecewise polynomials returned by CSINTERP (page 111), CSSHAPE (page 116), and CSSMOOTH (page 159)
- One-dimensional B-splines returned by BSINTERP (page 120), BSLSQ (page 144), and CONLSQ (page 154)
- Two-dimensional, tensor-product B-splines returned from BSINTERP (page 120) and BSLSQ (page 144)

If *spline* is a piecewise polynomial, function SPVALUE computes the values of a cubic spline or one of its derivatives. In this case, the user is required to supply the arguments *x* and *spline* and must not supply the argument *y*. If *x* is a scalar, then a scalar is returned. If *x* is a one-dimensional array, then a one-dimensional array of values is returned.The first and last pieces of the cubic spline are extrapolated so that the cubic spline structures returned by the cubic spline routines are defined and can be evaluated on the entire real line. This ability is based on the routine PPVALU by de Boor (1978, p. 89).

If *spline* is a one-dimensional B-spline, the SPVALUE function computes the values of a spline or one of its derivatives. In this case, the user is required to supply the arguments *x* and *spline* and must not supply the argument *y*. If *x* is a scalar, then a scalar is returned. If *x* is a one-dimensional array, then a one-dimensional array of values is returned. This ability is based on the routine BVALUE by de Boor (1978, p. 144).

If *spline* is a two-dimensional, tensor-product B-spline, the SPVALUE function computes the values of a tensor-product spline or one of its derivatives. In this case, the user is required to supply the arguments *x*, *y*, and *spline*. If *x* and *y* are both scalars, then a scalar is returned. If *x* and *y* are both one-dimensional arrays, then a two-dimensional array of values is returned, where the (*i*, *j*)-th element of the returned matrix is the desired value of SPLINE (*x* (*i*), *y* (*j*)). This ability is based on the discussion in de Boor (1978, pp. 351–353).

Example 1

In this example, a cubic spline interpolant to function values is computed. The spline is then evaluated, and the results are plotted. Since the default settings are used, the interpolant is determined by the not-a-knot condition (see de Boor 1978).

```
x = FINDGEN(10)/9
f = SIN(15 * x)
pp = CSINTERP(x, f)
x2 = FINDGEN(100)/99
ppeval = SPVALUE(x2, pp)
PLOT, x2, ppeval
```

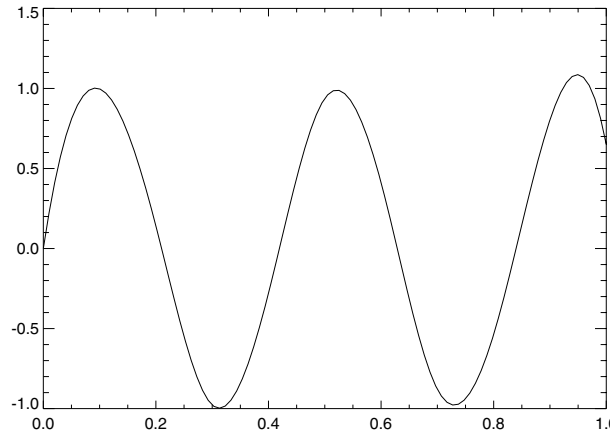


Figure 3-8 Plot of spline evaluation.

Example 2

This example computes a two-dimensional, tensor-product B-spline using BSINTERP (page 120), then uses SPVALUE to evaluate the spline on a grid, and plots the results.

```
x = FINDGEN(5)/4
y = FINDGEN(5)/4
f = FLTARR(5, 5)
FOR i = 0, 4 DO f(i,*) = SIN(2 * !Pi * x(i)) * (-COS(!Pi*y/2))
        ; Generate the data.
bs = BSINTERP(x, y, f)
    ; Compute the spline by calling BSINTERP.
```

```
bsval = FLTARR(20, 20)
FOR i = 0, 19 DO BSVAL(i, *) = SPVALUE(i/19., FINDGEN(20)/19,
   bs)
      ; Evaluate the spline on a grid.
!P.Multi = [0, 1, 2]
WINDOW, XSize = 400, YSize = 800
   ; Plot the original data and the evaluations of the spline in the same plot window.
ax = 50
   ; The angle of rotation about the x-axis in the plots is defined by ax.
!P.Charsize = 1.5
SURFACE, f, x, y, Ax = ax, XTitle = 'X', $
   YTitle = 'Y'
SURFACE, bsval, FINDGEN(20)/19, $
   FINDGEN(20)/19, Ax = ax, $
   XTitle = 'X', YTitle = 'Y'
```

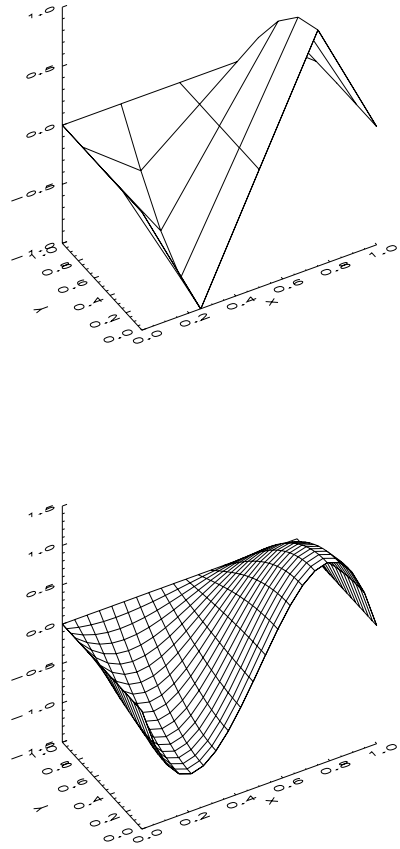


Figure 3-9 Plot of evaluations of two-dimensional spline.

Warning Errors

MATH_X_NOT_WITHIN_KNOTS — Value of *x* does not lie within the knot sequence.

MATH_Y_NOT_WITHIN_KNOTS — Value of *y* does not lie within the knot sequence.

Fatal Errors

MATH_KNOT_MULTIPLICITY — Multiplicity of the knots cannot exceed the order of the spline.

MATH_KNOT_NOT_INCREASING — Knots must be nondecreasing.

SPINTEG Function

Computes the integral of a one- or two-dimensional spline.

Usage

result = SPINTEG(*a*, *b*, *spline*)

result = SPINTEG(*a*, *b*, *c*, *d*, *spline*)

Input Parameters

If integration of a one-dimensional spline is desired, then arguments *a*, *b*, and *spline* are required. If integration of a two-dimensional spline is desired, then *a*, *b*, *c*, *d*, and *spline* are required.

a — Right endpoint of integration.

b — Left endpoint of integration.

c — Right endpoint of integration for the second variable of the tensor-product spline. This argument should only be used if *spline* is a two-dimensional, tensor-product spline.

d — Left endpoint of integration for the second variable of the tensor-product spline. This argument should only be used if *spline* is a two-dimensional, tensor-product spline.

spline — Structure that represents the spline to be integrated.

Returned Value

result — If *spline* is a one-dimensional spline, then the returned value is the integral from *a* to *b* of *spline*. If *spline* is a two-dimensional, tensor-product spline, then the returned value is the value of the integral of *spline* over the rectangle [*a*, *b*] x [*c*, *d*]. If no value can be computed, NaN (Not a Number) is returned.

Discussion

Function SPINTEG can be used to integrate splines of the following type:

- Piecewise polynomials returned by CSINTERP (page 111), CSSHAPE (page 116), and CSSMOOTH (page 159)

- One-dimensional B-splines returned by BSINTERP (page 120), BSLSQ (page 144), and CONLSQ (page 154)
- Two-dimensional, tensor-product B-splines returned from BSINTERP and BSLSQ

If *s* = *spline* is a one-dimensional piecewise polynomial or B-spline, then SPINTEG computes

$$\int_a^b s(x)dx .$$

If *spline* is a one-dimensional B-spline, then this function uses identity (22) of de Boor (1978, p. 115).

If *s* = *spline* is a two-dimensional, tensor-product spline, then the arguments *c* and *d* are required, and SPINTEG computes

$$\int_a^b \int_c^d s(x, y)dydx .$$

This function uses the (univariate integration) identity (22) of de Boor (1978, p. 151)

$$\int_{\mathbf{t}_0}^{x} \sum^{n-1} \alpha_i B_{i,k}(\tau)d\tau = \sum^{r-1} \left[\sum^{i} \alpha_j \frac{\mathbf{t}_{j+k} - \mathbf{t}_j}{k} \right] B_{i,k+1}(x)$$

where $\mathbf{t}_0 \le x \le \mathbf{t}_r$. It assumes (for all knot sequences) that the first and last *k* knots are stacked; that is, $\mathbf{t}_0 = \ldots = \mathbf{t}_{k-1}$ and $\mathbf{t}_n = \ldots = \mathbf{t}_{n+k-1}$, where *k* is the order of the spline in the *x* or *y* direction.

Example

In this example, a cubic spline interpolant to function values is computed. The values of the integral of this spline are then compared with the exact integral values. Since the default settings are being used, the interpolant is determined by the not-a-knot condition (de Boor 1978).

```
n = 21
    ; Generate the data.
x = FINDGEN(n)/(n - 1)
f = SIN(15 * x)
pp = CSINTERP(x, f)
    ; Compute the interpolant.
results = FLTARR(22, 4)
    ; Define an array to hold some results to be output later.
FOR i = n/2, 3 * n/2 DO BEGIN $
   x2 = i/FLOAT(2 * n - 2) &$
   y = SPINTEG(0, x2, pp) &$
   results(i - n/2, *) = &$
          [x2, (1 - COS(15 * x2))/15, y, &$
          ABS((1 - COS(15 * x2))/15 - y)] &$
       ; Loop over different limits of integration and compare the
       ; results with the true answer.
ENDFOR
PM, results, Format = '(4f12.4)', $
 Title  = '   X    True   Approx   Error'
       ; Output the results.
```

```
X          True          Approx        Error
0.2500        0.1214        0.1215        0.0001
0.2750        0.1036        0.1037        0.0001
0.3000        0.0807        0.0808        0.0001
0.3250        0.0559        0.0560        0.0001
0.3500        0.0325        0.0327        0.0001
0.3750        0.0139        0.0141        0.0002
0.4000        0.0027        0.0028        0.0002
0.4250        0.0003        0.0004        0.0002
0.4500        0.0071        0.0073        0.0002
0.4750        0.0223        0.0224        0.0001
0.5000        0.0436        0.0437        0.0001
```

```
0.5250        0.0681        0.0682        0.0001
0.5500        0.0924        0.0925        0.0001
0.5750        0.1131        0.1132        0.0001
0.6000        0.1274        0.1275        0.0001
0.6250        0.1333        0.1333        0.0001
0.6500        0.1298        0.1299        0.0001
0.6750        0.1176        0.1177        0.0001
0.7000        0.0984        0.0985        0.0001
0.7250        0.0747        0.0748        0.0001
0.7500        0.0499        0.0500        0.0001
0.7750        0.0274        0.0276        0.0001
```

Warning Errors

MATH_SPLINE_LEFT_ENDPT — Left endpoint of x integration is not within the knot sequence. Integration occurs only from $\mathbf{t}_{Order-1}$ to b.

MATH_SPLINE_RIGHT_ENDPT — Right endpoint of x integration is not within the knot sequence. Integration occurs only from $\mathbf{t}_{Order-1}$ to a.

MATH_SPLINE_LEFT_ENDPT_1 — Left endpoint of x integration is not within the knot sequence. Integration occurs only from b to $\mathbf{t}_{Spline\_Space\_Dim-1}$.

MATH_SPLINE_RIGHT_ENDPT_1 — Right endpoint of x integration is not within the knot sequence. Integration occurs only from a to $\mathbf{t}_{Spline\_Space\_Dim-1}$.

MATH_SPLINE_LEFT_ENDPT_2 — Left endpoint of y integration is not within the knot sequence. Integration occurs only from $\mathbf{t}_{Order-1}$ to d.

MATH_SPLINE_RIGHT_ENDPT_2 — Right endpoint of y integration is not within the knot sequence. Integration occurs only from $\mathbf{t}_{Order-1}$ to c.

MATH_SPLINE_LEFT_ENDPT_3 — Left endpoint of y integration is not within the knot sequence. Integration occurs only from d to $\mathbf{t}_{Spline\_Space\_Dim-1}$.

MATH_SPLINE_RIGHT_ENDPT_3 — Right endpoint of y integration is not within the knot sequence. Integration occurs only from c to $\mathbf{t}_{Spline\_Space\_Dim-1}$.

Fatal Errors

MATH_KNOT_MULTIPLICITY — Multiplicity of the knots cannot exceed the order of the spline.

MATH_KNOT_NOT_INCREASING — Knots must be nondecreasing.

FCNLSQ Function

Computes a least-squares fit using user-supplied functions.

Usage

result = FCNLSQ(*f*, *nbasis*, *xdata*, *fdata*)

Input Parameters

f — Scalar string specifying the name of a user-supplied function that defines the subspace from which the least-squares fit is to be performed. The *k*-th basis function evaluated at *x* is *f* (*k*, *x*), where *k* = 1, 2, …, *nbasis*.

nbasis — Number of basis functions.

xdata — One-dimensional array containing the abscissas of the least-squares problem.

fdata — One-dimensional array containing the ordinates of the least-squares problem.

Returned Value

result — A one-dimensional array containing the coefficients of the basis functions.

Input Keywords

Double — If present and nonzero, double precision is used.

Weights — Array of weights used in the least-squares fit.

Output Keywords

Intercept — Named variable into which the coefficient of a constant function used to augment the user-supplied basis functions in the least-squares fit is stored. Setting this keyword forces an intercept to be added to the model.

SSE — Named variable into which the error sum of squares is stored.

Discussion

For a discussion of the principles relating to the implementation of FCNLSQ, refer to the discussion of FCNLSQ in the *PV-WAVE: IMSL Mathematics Reference*.

Function FCNLSQ computes a best least-squares approximation to given univariate data of the form

$$\{(x_i, f_i)\}_{i=0}^{n-1}$$

by M basis functions

$$\{F_j\}_{j=1}^{M}$$

(where M = *nbasis*). In particular, the default for this function returns the coefficients a which minimize

$$\sum_{i=0}^{n-1} w_i \left(f_i - \sum_{j=1}^{M} a_{j-1} F_j(x_i) \right)^2$$

where w = *Weights*, n = N_ELEMENTS (*xdata*), x = *xdata*, and f = *fdata*.

If optional argument *Intcercept* is chosen, then an intercept is placed in the model and the coefficients a, returned by FCNLSQ, minimize the error sum of squares as indicated below.

$$\sum^{n-1} w_i \left(f_i - \text{intercept} - \sum^{M} a_{j-1} F_j(x_i) \right)^2$$

Example

In this example, the following function is fit:

$$1 + \sin x + 7\sin 3x$$

This function is evaluated at 90 equally spaced points on the interval [0, 6]. Four basis functions, 1, sin$x$, sin2x, and sin3x, are used.

```
.RUN
    ; Define the basis functions.
- FUNCTION f, k, x
- IF (k EQ 1) THEN RETURN, 1. $
```

```
-   ELSE RETURN, SIN((k - 1) * x)
- END
% COMPILED module: F.
n = 90
xdata = 6 * FINDGEN(n)/(n - 1)
fdata = 1 + SIN(xdata) + 7 * SIN(3 * xdata)
nbasis = 4
    ; Generate the data.
coefs = FCNLSQ("f", nbasis, xdata, fdata)
    ; Compute the coefficients summing FCNLSQ.
PM, coefs, Format = '(f10.5)'
    ; Print the results.
   1.00000
   1.00000
   0.00000
   7.00000
```

Warning Errors

MATH_LINEAR_DEPENDENCE — Linear dependence of the basis functions exists. One or more components of *coef* are set to zero.

MATH_LINEAR_DEPENDENCE_CONST — Linear dependence of the constant function and basis functions exists. One or more components of *coef* are set to zero.

Fatal Errors

MATH_NEGATIVE_WEIGHTS_2 — All weights must be greater than or equal to zero.

BSLSQ Function

Computes a one- or two-dimensional, least-squares spline approximation.

Usage

result = BSLSQ(*xdata*, *fdata*, *xspacedim*)

result = BSLSQ(*xdata*, *ydata*, *fdata*, *xspacedim*, *yspacedim*)

Input Parameters

If a one-dimensional B-spline is desired, then arguments *xdata*, *fdata*, and *xspacedim* are required. If a two-dimensional, tensor-product B-spline is desired, then arguments *xdata*, *ydata*, *fdata*, *xspacedim*, and *yspacedim* are required.

xdata — One-dimensional array containing the data points in the *x*-direction.

ydata — One-dimensional array containing the data points in the *y*-direction.

fdata — Array containing the values to be approximated. If a one-dimensional approximation is to be computed, then *fdata* is a one-dimensional array. If a two-dimensional approximation is to be computed, then *fdata* is a two-dimensional array, where *fdata* (*i*, *j*) contains the value at (*xdata* (*i*), *ydata*(*j*)).

xspacedim — Linear dimension of the spline subspace for the *x* variable. It should be smaller than the number of data points in the *x*-direction and greater than or equal to the order of the spline in the *x*-direction (whose default value is 4).

yspacedim — Linear dimension of the spline subspace for the *y* variable. It should be smaller than the number of data points in the *y*-direction and greater than or equal to the order of the spline in the *y*-direction (whose default value is 4).

Returned Value

result — A structure containing all the information to determine the spline fit.

Input Keywords

Double — If present and nonzero, double precision is used.

XOrder — Specifies the order of the spline in the *x*-direction.

Default: *XOrder* = 4, i.e., cubic splines

YOrder — Specifies the order of the spline in the *y*-direction. If a one-dimensional spline is being computed, then *YOrder* has no effect on the computations.

Default: *YOrder* = 4, i.e., cubic splines

XKnots — Specifies the array of knots in the *x*-direction to be used when computing the definition of the spline.

Default: knots are equally spaced in the *x*-direction

YKnots — Specifies the array of knots in the *y*-direction to be used when computing the definition of the spline.

Default: knots are equally spaced in the *y*-direction

XWeights — Array containing the weights to use in the *x*-direction.

Default: all weights equal to 1

YWeights — Array containing the weights to use in the *y*-direction. If a one-dimensional spline is being computed, then *YWeights* has no effect on the computations.

Default: all weights equal to 1

Optimize — If present and nonzero, optimizes the knot locations by attempting to minimize the least-squares error as a function of the knots. This keyword is only active if a one-dimensional spline is being computed.

Output Keywords

Sse — Specifies the named variable into which the weighted error sum of squares is stored.

Discussion

Function BSLSQ computes a least-squares approximation to weighted data returning either a one-dimensional B-spline or a two-dimensional, tensor-product B-spline. The determination of whether to return a one- or two-dimensional spline is made based on the number of arguments passed to the function.

One-dimensional, B-spline Least-squares Approximations

Make the following identifications:

n = N_ELEMENTS(*xdata*)

x = *xdata*

f = *fdata*

m = *xspacedim*

k = *XOrder*

For convenience, assume that the sequence x is increasing (although the function does not require this).

By default, $k = 4$ and the knot sequence selected equally distributes the knots through the distinct x_i's. In particular, the $m + k$ knots are generated in $[x_0 , x_{n-1}]$ with k knots stacked at each of the extreme values. The interior knots are equally spaced in the interval.

Once knots **t** and weights w are determined (and assuming that keyword *Optimize* is not set), then the function computes the spline least-squares fit to the data by minimizing over the linear coefficients a_j, such that

$$\sum_{i=0}^{n-1} w_i \left(f_i - \sum_{j=0}^{m-1} a_j B_j(x_i) \right)^2$$

where B_j, $j = 0, \ldots, m-1$, is a (B-spline) basis for the spline subspace.

The *XOrder* keyword allows the user to choose the order of the spline fit. The *XKnots* keyword allows user specification of knots. The one-dimensional functionality of BSLSQ is based on the routine L2APPR by de Boor (1978, p. 255).

If option *Optimize* is chosen, the function attempts to find the best placement of knots that minimizes the least-squares error to the given data by a spline of order k with m coefficients. For this problem to make sense, it is necessary that $m > k$. Then, to find the minimum of the functional, use the following:

$$F(a, \mathbf{t}) = \sum_{i=0}^{n-1} w_i \left(f_i - \sum_{j=0}^{m-1} a_j B_{j,k,\mathbf{t}}(x_i) \right)^2$$

The technique employed here uses the fact that for a fixed-knot sequence **t** the minimization in *a* is a linear least-squares problem that can be easily solved. Thus, the objective function *F* is a function of only **t** by setting the following:

$$G(\mathbf{t}) = \min F(a, \mathbf{t})$$

A Gauss-Seidel (cyclic coordinate) method is then used to reduce the value of the new objective function *G*. In addition to this local method, there is a global heuristic built into the algorithm that is useful if the data arise from a smooth function. This heuristic is based on the routine NEWNOT of de Boor (1978, pp. 184, 258–261).

The initial guess, $\mathbf{t}^g$, for the knot sequence is either provided by the user or is the default. This guess must be a *valid* knot sequence for splines of order *k* with

$$\mathbf{t}^g_0 \le \ldots \le \mathbf{t}^g_{k-1} \le x_i \le \mathbf{t}^g_m \le \ldots \le \mathbf{t}^g_{m+k-1} \qquad i = 1, \ldots, M$$

with $\mathbf{t}^g$ nondecreasing and

$\mathbf{t}^g_i < \mathbf{t}^g_{i+k}$ for $i = 0, \ldots, m-1$.

In regard to execution speed, this function can be several orders of magnitude slower than a simple least-squares fit.

The return value for this function is a structure containing all the information to determine the spline (stored in B-spline form) that is computed by this function.

In the figure below, two cubic splines are fit to SQRT($|x|$) . Both splines are cubics with the same *xspacedim* = 8. The first spline is computed with the default settings, while the second spline is computed by optimizing the knot locations using the *Optimize* keyword.

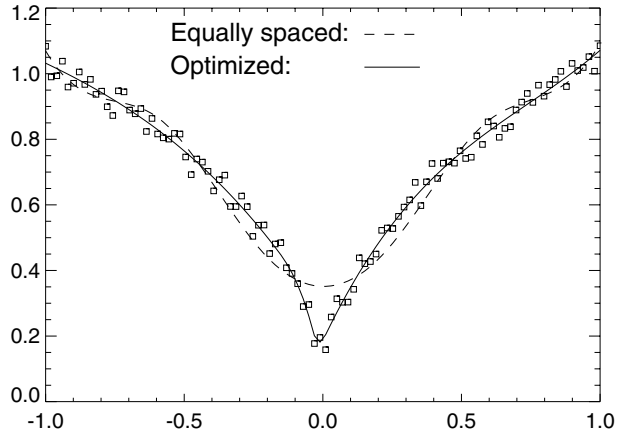


Figure 3-10 Two fits to noisy SQRT(|x|).

Two-dimensional, B-spline Least-squares Approximations

If a two-dimensional, tensor-product B-spline is desired, the BSLSQ function computes a tensor-product spline, least-squares approximation to weighted, tensor-product data. The input for this function consists of data vectors to specify the tensor-product grid for the data, two vectors with the weights (optional, the default is 1), the values of the surface on the grid, and the specification for the tensor-product spline (optional, a default is chosen). The grid is specified by the two vectors $x = xdata$ and $y = ydata$ of length n = N_ELEMENTS(*xdata*) and m= N_ELEMENTS(*ydata*), respectively. A two-dimensional array $f = fdata$ contains the data values to be fit. The two vectors $w_x = XWeights$ and $w_y = YWeights$ contain the weights for the weighted, least-squares problem. The information for the approximating tensor-product spline can be provided using keywords *XOrder*, *YOrder*, *XKnots*, and *YKnots*. This information is contained in $k_x = XOrder$, $\mathbf{t}_x = XKnots$, and $n = xspacedim$ for the spline in the first variable, and in $k_y = YOrder$, $\mathbf{t}_y = YKnots$, and $m = yspacedim$ for the spline in the second variable.

This function computes coefficients for the tensor-product spline by solving the normal equations in tensor-product form as discussed in de Boor (1978, Chapter 17). For more information, see the paper by Grosse (1980).

As the computation proceeds, coefficients c are obtained minimizing

$$\left(\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} w_x(i) w_y(j) \left[\sum_{k=0}^{N-1} \sum_{l=0}^{M-1} c_{kl} B_{kl}(x_i, y_i) - f_{ij} \right]^2 \right)$$

where the function B_{kl} is the tensor-product of two B-splines of order k_x and k_y:

$$B_{kl}(x, y) = B_{k, k_x, \mathbf{t}_x}(x) B_{l, k_y, \mathbf{t}_y}(y)$$

The spline

$$\sum_{N-1} \sum_{M-1} c_{kl} B_{kl}$$

and its partial derivatives can be evaluated using function SPVALUE (page 132).

The return value for this function is a structure containing all the information to determine the spline that is computed by this function. For example, the following code sequence evaluates this spline (stored in the structure *sp*) at (*x*, *y*) and returns the value in v:

```
v = SPVALUE(x, y, sp)
```

Example 1

This example fits data generated from a trigonometric polynomial

$$1 + sinx + 7sin3x + \varepsilon$$

where ε is a random uniform deviate over the range [–1, 1]. The data are obtained by evaluating this function at 90 equally spaced points on the interval [0, 6]. This data is fit with a cubic spline with 12 degrees of freedom (eight equally spaced interior knots). The computed fit and original data are then plotted as follows:

```
n = 90
x = 6 * FINDGEN(n)/(n - 1)
f = 1 + SIN(x) + 7 * SIN(3 * x) + $
   (1 - 2 * RANDOM(n))
```

```
        ; Set up the data.
sp = BSLSQ(x, f, 8)
    ; Compute the spline fit.
speval = SPVALUE(x, sp)
    ; Evaluate the computed spline at the original data abscissa.
PLOT, x, speval
    ; Plot the results.
OPLOT, x, f, Psym = 6
```

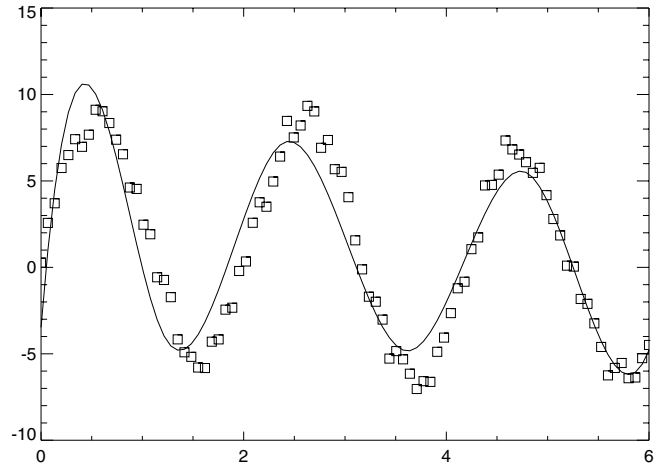


Figure 3-11 One-dimensional least-squares B-spline fit.

Example 2

This example fits noisy data that arises from the function $e^x \sin(x + y) + \varepsilon$, where ε is a random uniform deviate in the range (–1, 1), on the rectangle [0, 3] x [0, 5]. This function is sampled on a 50 x 25 grid and the original data and the evaluations of the computed spline are plotted.

```
nx = 50
ny = 25
    ; Generate noisy data on a 50 x 25 grid.
x = 3 * FINDGEN(nx)/(nx - 1)
y = 5 * FINDGEN(ny)/(ny - 1)
f = FLTARR(nx, ny)
FOR i = 0, nx - 1 DO f(i, *) = EXP(x(i)) * $
   SIN(x(i) + y) + 2 * RANDOM(ny) - 1
```

```
sp = BSLSQ(x, y, f, 5, 7)
    ; Call BSLSQ to compute the least-squares fit. Notice that
    ; xspacedim = 5 and yspacedim = 7.
speval = SPVALUE(x, y, sp)
    ; Evaluate the fit on the original grid.
!P.Multi = [0, 1, 2]
WINDOW, XSize = 500, YSize = 800
    ; Plot the original data and the fit in the same window.
SURFACE, f, x, y, Ax = 45, $
   XTitle = 'X', YTitle = 'Y'
SURFACE, speval, x, y, Ax = 45, $
   XTitle = 'X', YTitle = 'Y'
```

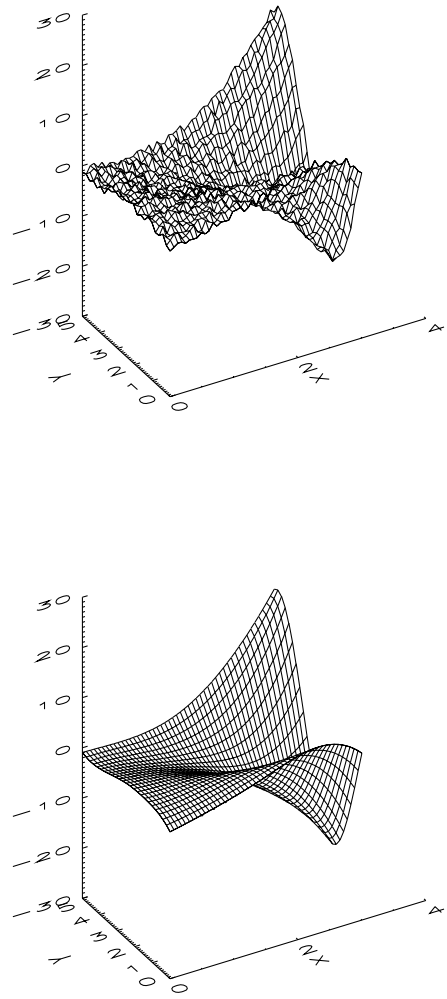


Figure 3-12 Two-dimensional B-spline least-squares fit to noisy data.

Warning Errors

MATH_ILL_COND_LSQ_PROB — Least-squares matrix is ill-conditioned. Solution might not be accurate.

MATH_SPLINE_LOW_ACCURACY — There may be less than one digit of accuracy in the least-squares fit. Try using higher precision if possible.

MATH_OPT_KNOTS_STACKED_1 — Knots found to be optimal are stacked more than *Order*. This indicates that fewer knots will produce the same error sum of squares. Knots have been separated slightly.

Fatal Errors

MATH_KNOT_MULTIPLICITY — Multiplicity of the knots cannot exceed the order of the spline.

MATH_KNOT_NOT_INCREASING — Knots must be nondecreasing.

MATH_SPLINE_LRGST_ELEMNT — Data arrays *xdata* and *ydata* must satisfy $data_i \leq \mathbf{t}_{Spline\_Space\_Dim}$, for $i = 1, \ldots, num\_data$.

MATH_SPLINE_SMLST_ELEMNT — Data arrays *xdata* and *ydata* must satisfy $data_i \geq \mathbf{t}_{Order-1}$, for $i = 1, \ldots, num\_data$.

MATH_NEGATIVE_WEIGHTS — All weights must be greater than or equal to zero.

MATH_DATA_DECREASING — The *xdata* values must be nondecreasing.

MATH_XDATA_TOO_LARGE — Array *xdata* must satisfy $xdata_i \leq \mathbf{t}_{ndata}$, for $i = 1, \ldots, ndata$.

MATH_XDATA_TOO_SMALL — Array *xdata* must satisfy $xdata_i \geq \mathbf{t}_{Order-1}$, for $i = 1, \ldots, ndata$.

MATH_OPT_KNOTS_STACKED_2 — Knots found to be optimal are stacked more than *Order*. This indicates fewer knots will produce the same error sum of squares.

CONLSQ Function

Computes a least-squares constrained spline approximation.

Usage

result = CONLSQ(*xdata, fdata*, *spacedim*, *constraints* [, *nhard*])

Input Parameters

xdata — One-dimensional array containing the abscissas of the least-squares problem.

fdata — One-dimensional array containing the ordinates of the least-squares problem.

spacedim — Linear dimension of the spline subspace. It should be smaller than the number of data points and greater than or equal to the order of the spline (whose default value is 4).

constraints — Array of structures containing the abscissas at which the fit is to be constrained, the derivative of the spline that is to be constrained, the type of constraints, and any lower or upper limits. A description of the structure fields is in the table below.

Tag	Description
XVAL	Point at which fit is constrained (float)
DER	Derivative value of the spline to be constrained (long int)
TYPE	Types of the general constraints (long int)
BL	Lower limit of the general constraints (float)
BU	Upper limit of the general constraints (float)

NOTE To constrain the integral of the spline over the closed interval (c, d), set *constraints* (i).XVAL = c and *constraints* (i + 1).XVAL = d. For consistency, insist that *constraints* (i).TYPE = *constraints* (i + 1).TYPE = 5, 6, 7, or 8 and $c \leq d$.

For more information on the allowed values of *constraints*.TYPE, refer to the description of CONLSQ.

constraints(i).TYPE	*i-th constraint*
1	$bl_i = f^{(d_i)}(x_i)$
2	$f^{(d_i)}(x_i) \le bu_i$
3	$f^{(d_i)}(x_i) \ge bl_i$
4	$bl_i \le f^{(d_i)}(x_i) \le bu_i$
5	$bl_i = \int_c^d f(t)dt$
6	$\int_c^d f(t)dt \le bu_i$
7	$\int_c^d f(t)dt \ge bl_i$
8	$bl_i \le \int_c^d f(t)dt \le bu_i$
20	periodic end conditions
99	disregard this constraint

In order to have two-point constraints,
constraints(*i*).TYPE = *constraints*(*i* + 1).TYPE is needed.

constraints(i).TYPE	*i-th constraint*
9	$bl_i = f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1})$
10	$f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \le bu_i$
11	$f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \ge bl_i$
12	$bl_i \le f^{(d_i)}(x_i) - f^{(d_{i+1})}(x_{i+1}) \le bu_i$

nhard — (Optional) Number of entries of *constraints* involved in the "hard" constraints. Note that $0 \le nhard \le$ (SIZE (*constraints*)) (1). The default, *nhard* = 0, always results in a fit, while setting *nhard* = (SIZE (*constraints*)) (1) forces all constraints to be met. The "hard" constraints must be met or the function signals fail. The "soft" constraints need not be satisfied, but there is an attempt to satisfy the "soft" constraints. The constraints must be listed in terms of priority with the most important constraints first. Thus, all "hard" constraints must precede "soft" constraints. If infeasibility is detected among the "soft" constraints, the function satisfies, in order, as many of the "soft" constraints as possible.

Default: *nhard* = 0

Returned Value

result — A structure that represents the spline fit.

Input Keywords

Double — If present and nonzero, double precision is used.

Order — Specifies the order of the spline.

Default: *Order* = 4, i.e., cubic splines

Knots — Specifies the array of knots to be used when computing the spline.

Default: knots are equally spaced

Weights — Array containing the weights to be used.

Default: all weights equal 1

Discussion

Function CONLSQ produces a constrained, weighted, least-squares fit to data from a spline subspace. Constraints involving one-point, two-points, or integrals over an interval are allowed.

The types of constraints supported by the functions are of four types:

$$E_p[f] = f^{(j_p)}(y_p)$$

$$\text{or} = f^{(j_p)}(y_p) - f^{(j_{p+1})}(y_{p+1})$$

$$\text{or} = \int_{y_p}^{y_{p+1}} f(t)dt$$

$$\text{or} = \text{periodic end conditions}$$

An interval, I_p, (which may be a point, a finite interval, or a semi-infinite interval) is associated with each of these constraints.

The input for this function consists of the data set (x_i, f_i) for $i = 1, ..., N$ (where N = N_ELEMENTS(*xdata*)); that is, the data which is to be fit and the dimension of the spline space from which a fit is to be computed, *spacedim*. The *constraints* argument is an array of structures that contains the abscissas of the points involved in specifying the constraints, as well as information relating the type of constraints and the constraint interval. The optional argument *nhard*

allows users of this code to specify which constraints must be met and which constraints can be removed in order to compute a fit. The algorithm tries to satisfy all the constraints, but if the constraints are inconsistent, then it drops constraints in the reverse order specified, until either a consistent set of constraints is found or the "hard" constraints are determined to be inconsistent (the "hard" constraints are those involving *constraints*(0), ..., *constraints*(*nhard* – 1)).

Let n_f denote the number of feasible constraints as described above. The function solved the problem

$$\sum^{n} \left| f_i - \sum^{m} a_j B_j(x_i) \right|^2 w_i$$

subject to

$$E_p\left[\sum^{m} a_j B_j\right] \in I_p \quad p = 1, \ldots, n_f$$

This linearly constrained least-squares problem is treated as a quadratic program and is solved by invoking function QUADPROG (page 335).

The choice of weights depends on the data uncertainty in the problem. In some cases, there is a natural choice for the weights based on the estimates of errors in the data points.

Determining feasibility of linear constraints is a numerically sensitive task. If difficulties are encountered, a quick fix is to widen the constraint intervals I_p.

Example

This example is a simple application of CONLSQ. Data from the function $x/2 + \sin(x/2)$ contaminated with random noise is generated and then fit with cubic splines. The function is increasing so it is hoped that the least-squares fit also is increasing. This is not the case for the unconstrained least-squares fit generated by function BSLSQ (page 144). The derivative is then forced to be greater than zero at 15 equally spaced points and CONLSQ is called. The resulting curve is monotone.

```
RANDOMOPT, Set = 234579
    ; Set the random seed.
ndata = 15;
```

```
spacedim = 8;
    ; Generate the data to be fit.
x = 10 * FINDGEN(ndata)/(ndata - 1)
y = .5 * (x) + SIN(.5 * (x)) + RANDOM(ndata) - .5
sp1 = BSLSQ(x, y, spacedim)
    ; Compute the unconstrained least-squares fit.
nconstraints = 15
    ; Define the constraints to be used by CONLSQ.
constraints = REPLICATE({constraint, $
   XVAL:0.0, DER:0L, TYPE:0L, BL:0.0, $
   BU:0.0}, nconstraints)
       ; Define an array of constraint structures. Each element of the
       ; array contains one structure that defines a constraint.
constraints.XVAL = 10*FINDGEN(nconstraints)/(nconstraints-1)
       ; Put a constant at 15 equally spaced points.
FOR i = 0, nconstraints - 1 DO BEGIN &$
   constraints(i).DER  = 1 &$
   constraints(i).TYPE = 3 &$
   constraints(i).BL   = 0. &$
ENDFOR
    ; Define the constraints to force the second derivative to be greater
    ; than zero at the 15 equally spaced points.
sp2 = CONLSQ(x, y, spacedim, constraints)
    ; Call CONLSQ.
nplot = 100
xplot = 10 * FINDGEN(nplot)/(nplot - 1)
yplot1 = SPVALUE(xplot, sp1)
yplot2 = SPVALUE(xplot, sp2)
PLOT, xplot, yplot1, Linestyle = 2
    ; Plot the results.
OPLOT, xplot, yplot2
OPLOT, x, y, Psym = 6
XYOUTS, 1, 4.5, 'CONLSQ', Charsize = 2
XYOUTS, 1, 4, 'BSLSQ', Charsize = 2
OPLOT, [3.6, 4.6], [4.6, 4.6]
OPLOT, [3.6, 4.6], [4.1, 4.1], Linestyle = 2
```

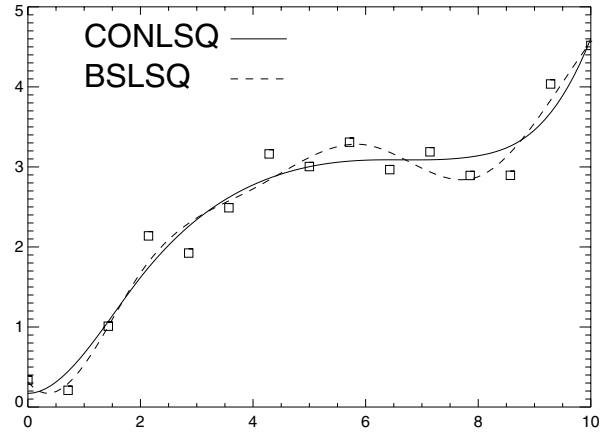


Figure 3-13 Monotonic B-spline fit to noisy data.

CSSMOOTH Function

Computes a smooth cubic spline approximation to noisy data by using cross-validation to estimate the smoothing parameter or by directly choosing the smoothing parameter.

Usage

result = CSSMOOTH(*xdata*, *fdata*)

Input Parameters

xdata — One-dimensional array containing the abscissas of the problem.

fdata — One-dimensional array containing the ordinates of the problem.

Returned Value

result — The structure that represents the cubic spline.

Input Keywords

Double — If present and nonzero, double precision is used.

Weights — Array containing the weights to be used in the problem.

Default: all weights are equal to 1

Smpar — Specifies the real, scalar smoothing parameter explicitly. See *Discussion* below for more details.

Discussion

Function CSSMOOTH is designed to produce a C^2 cubic spline approximation to a data set in which the function values are noisy. This spline is called a *smoothing spline*.

Consider first the situation when the optional keyword *Smpar* is selected. Then, a natural cubic spline with knots at all the data abscissas x = *xdata* is computed, but it does *not* interpolate the data (x_i, f_i). The smoothing spline s is the unique C^2 function which minimizes

$$\int_a^b s''(x)^2 dx$$

subject to the constraint

$$\sum_{i=0}^{n-1} \left|(s(x_i) - f_i)w_i\right|^2 \leq \sigma$$

where w = *Weights*, σ = *Smpar* is the smoothing parameter, and n = N_ELEMENTS(*xdata*).

Recommended values for σ depend on the weights w. If an estimate for the standard deviation of the error in the value f_i is available, then w_i should be set to the inverse of this value. The smoothing parameter σ should be chosen in the confidence interval corresponding to the left side of the above inequality; that is

$$(n - \sqrt{2n} \leq \sigma \leq n + \sqrt{2n})$$

Function CSSMOOTH is based on an algorithm of Reinsch (1967). This algorithm also is discussed in de Boor (1978, pp. 235–243).

The default for this function chooses the smoothing parameter σ by a statistical technique called cross-validation. For more information on this topic, refer to Craven and Wahba (1979).

The return value for this function is a structure containing all the information to determine the spline (stored as a piecewise polynomial) that is computed by this procedure.

Example

In this example, function values are contaminated by adding a small “random” amount to the correct values. Function CSSMOOTH is used to approximate the original, uncontaminated data.

```
n = 25
x = 6 * FINDGEN(n)/(n - 1)
f = SIN(x) + .5 * (RANDOM(n) - .5)
    ; Generate the data.
pp = CSSMOOTH(x, f)
    ; Compute the fit.
x2 = 6 * FINDGEN(100)/99
    ; Evaluate the computed fit at 100 values in [0, 6].
ppeval = SPVALUE(x2, pp)
PLOT, x2, ppeval
    ; Plot the results.
OPLOT, x, f, Psym = 6, Symsize = .5
```

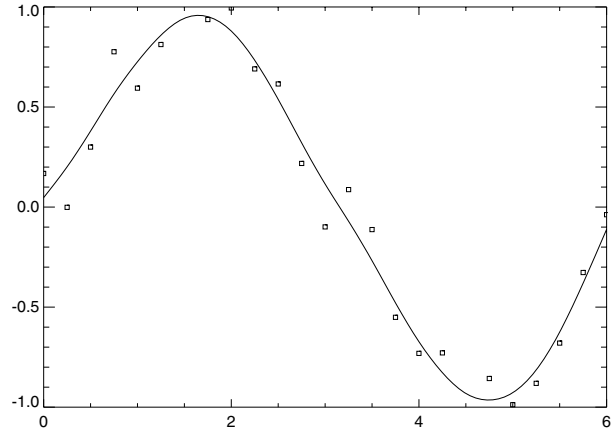


Figure 3-14 Smoothing spline.

Warning Errors

MATH_MAX_ITERATIONS_REACHED — Maximum number of iterations has been reached. The best approximation is returned.

Fatal Errors

MATH_DUPLICATE_XDATA_VALUES — The *xdata* values must be distinct.

MATH_NEGATIVE_WEIGHTS — All weights must be greater than or equal to zero.

WgSplineTool Procedure

Graphical User Interface (GUI) that computes a one-dimensional, least-squares spline fit to (possibly noisy) data.

Usage

WgSplineTool [, *fit*, *parent*, *shell*]

Input Parameters

parent — Widget handle of a parent widget. If supplied, it will be used as the parent shell for the tool.

Output Parameters

fit — Structure containing the computed spline. See the introduction of this chapter for a complete description of the spline structure.

shell — Used to pass the shell widget handle for the tool back to the caller.

Input Keywords

XData — One-dimensional array containing the abscissas of the data to be fit. If *XData* is specified, then *YData* must also be specified.

YData — One-dimensional array containing the ordinates of the data to be fit.

XSize — Width, in pixels, of the draw widget used to display the fit. Keyword *XSize* must be between 100 and 800.

Default: *XSize* = 600

YSize — Height, in pixels, of the draw widget used to display the fit. Keyword *YSize* must be between 100 and 800.

Default: *YSize* = 600

Demo_Data — If present and nonzero, sets path to the PV-WAVE example data directory for importing data. Otherwise, WwFileSelection prompts for input from the same directory from which WgSplineTool was invoked.

Foreground — Specifies the default foreground color name.

Background — Specifies the default background color name.

Font — Specifies the name of the default font used for text.

Position — Specifies the position of the upper-left corner of the main window on the screen.

Title — String specifying a title for the shell. The default is "Constrained Spline Tool."

Discussion

Procedure WgSplineTool is designed to help compute a least-squares spline fit to (possibly noisy) data. The WgSplineTool procedure computes the fit using function CONLSQ (page 154). The default use of WgSplineTool does not require any arguments or keywords and can be used as follows:

```
WgSplineTool
```

There are two ways to supply data to WgSplineTool. First, if the data is contained in variables, for example, *x* and *y*, invoke WgSplineTool with the following call:

```
WgSplineTool, XData = x, YData = y
```

The variables *x* and *y* should be one-dimensional arrays of equal length.

The *YData* keyword can be used without the *XData* keyword. in this case, it is assumed that the associated *x* values are monotonically increasing from 0. The call would be as follows:

```
WgSplineTool, YData = y
```

The variable *y* should be a one-dimensional array.

The second method to import data into the application is through an external ASCII file, along with the button labeled *Open*. The structure of the file is as follows:

- The first line contains the total number of (x, y) pairs.
- Each line after the first line should contain one (x, y) pair.

For example, the following six lines could be put into a file for use by WgSplineTool:

```
5
0.0454      3.43245
0.1243      4.45645
0.2454      1.43553
1.2311      1.76543
2.4325      9.53234
```

Once the data has been imported to the application, the data is automatically fitted with the default choice of knots. After this is accomplished, you have the ability to add the order of the fit, the spline space dimension, and the placement of the knots. To adjust the placement of the knots, simply use the mouse to "click and drag" the knots on the plot (the knot abscissae are the triangles that appear near the bottom of the plot). WgSplineTool returns the last computed fit in the argument *fit*.

Use the *Save Spline* button to save any intermediate fits. After saving the fit, you can restore the spline at a later time into a variable using the following command:

```
RESTORE, 'filename'
```

A description of each element of the top-level widget follows. The numbers correspond to the labels in *Figure 3-15*:

1. **Plotting Options:**
   Once a fit has been computed, the following plots can be viewed:
   - The computed fit.
   - The first derivative of the computed fit.
   - The second derivative of the computed fit.
2. **1st Derivative Constraints:**
   Once a fit has been computed, the following choices for global constraints on the first derivative of the computed fit can be enforced:
   - Does not enforce any constraints.
   - Forces the fit to be nondecreasing, i.e., the first derivative is non-negative throughout the domain of the data.
   - Forces the fit to be nonincreasing, i.e., the first derivative is nonpositive throughout the domain of the data.

3. **2nd Derivative Constraints:**
   Once a fit has been computed, the following choices for global constraints on the second derivative of the computed fit can be enforced:
   - Does not enforce any constraints.
   - Forces the fit to be convex, i.e., the second derivative is non-negative throughout the domain of the data.
   - Forces the fit to be concave, i.e., the second derivative is nonpositive throughout the domain of the data.

4. **Spline Parameters:**
   Once a fit has been computed, the following spline parameters can be changed using sliders:
   - *Spline Order*. Note: The order of the spline is one more than the degree of the fit.
   - *Spline Space Dimension.* Changing this value increases the number of knots used in computing the fit.

5. **Plot Window:**
   Displays a plot of the fit in this draw widget. Using the mouse, the knots can be moved from within this window.

6. **Mouse Coordinates:**
   As the mouse is moved over the plot, reflects the mouse coordinates. The coordinates are computed in data coordinates.

7. **Default Knots:**
   Fits the current data with a default set of knots.

8. **Open:**
   Imports data from an external ASCII file.

9. **Save Spline:**
   Allows saving of computed splines in save files.

10. **Dismiss:**
    Terminates this program.

11. **Help:**
    Opens the online help file for viewing.

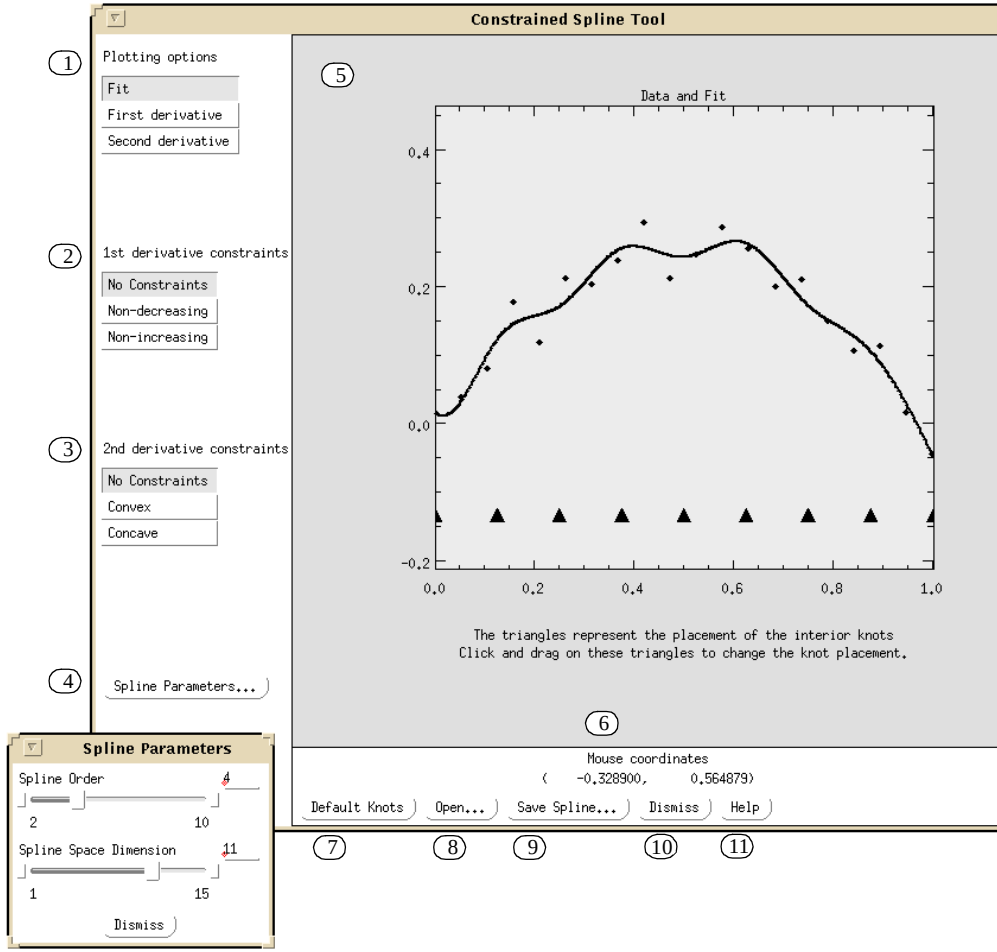


Figure 3-15 Spline fitting widget.

SMOOTHDATA1D Function

Smooths one-dimensional data by error detection.

Usage

result = SMOOTHDATA1D(*x*, *y*)

Input Parameters

x — One-dimensional array containing the abscissas of the data points.

y — One-dimensional array containing the ordinates of the data points.

Returned Value

result — One-dimensional array containing the smoothed data.

Input Keywords

Double — If present and nonzero, double precision is used.

Itmax — The maximum number of iterations allowed.

Default: *Itmax* = 500

Sc — The stopping criterion. *Sc* should be greater than or equal to zero.

Default: *Sc* = 0.0

Distance — Proportion of the distance the ordinate in error is moved to its interpolating curve. It must be in the range 0.0 to 1.0.

Default: *Distance* = 1.0

Algorithm

The function SMOOTHDATA1D is designed to smooth a data set that is mildly contaminated with isolated errors. In general, the routine will not work well if more than 25% of the data points are in error. The routine SMOOTHDATA1D is based on an algorithm of Guerra and Tapia (1974).

Setting N_ELEMENTS(*x*) = *n*, *Y* = *f*, *result* = *s* and *X*= *x*, the algorithm proceeds as follows. Although the user need not an ordered *x* sequence, we will

assume that x is increasing for simplicity. The algorithm first sorts the x values into an increasing sequence and then continues. A cubic spline interpolant is computed for each of the 6-point data sets (initially setting $s = f$)

$$(x_j, s_j) \qquad j = i - 3, \ldots, i + 3 \quad j \neq i,$$

where $i = 4, \ldots, n - 3$. For each i the interpolant, which we will call S_i, is compared with the current value of s_i, and a 'point energy' is computed as

$$pe_i = S_i(x_i) - s_i$$

Setting $sc = Sc$, the algorithm terminates either if *Itmax* iterations have taken place or if

$$|pe_i| \leq sc\frac{(x_{i+3} - x_{i-3})}{6} \qquad i = 4, \ldots, n-3$$

If the above inequality is violated for any i, then we update the i-th element of s by setting $s_i = s_i + d(pe_i)$, *where* $d = Distance$. Note that neither the first three nor the last three data points are changed. Thus, if these points are inaccurate, care must be taken to interpret the results.

The choice of the parameters *Distance*, *Sc* and *Itmax* are crucial to the successful usage of this subroutine. If the user has specific information about the extent of the contamination, then he should choose the parameters as follows: *Distance* = 1, $Sc = 0$ and *Itmax* to be the number of data points in error. On the other hand, if no such specific information is available, then choose *Distance* = .5, $Itmax \leq 2n$, and

$$Sc = 0.5\frac{\max s - \min s}{(x_n - x_1)}$$

In any case, we would encourage the user to experiment with these values.

Example

We take 91 uniform samples from the function $5 + (5 + t^2 \sin t)/t$ on the interval [1, 10]. Then, we contaminate 10 of the samples and try to recover the original function values.

```
FUNCTION F, xdata
    RETURN, (xdata*xdata*SIN(xdata) + 5)/xdata + 5
```

```
END

isub   =   [5, 16, 25, 33, 41, 48, 55, 61, 74, 82]

rnoise   =   [2.5, -3.0, -2.0, 2.5, 3.0, -2.0, -2.5, 2.0, -2.0,
   3.0]

   ; Example 1: No specific information available.

dis   =   0.5

sc   =   0.56

itmax   =   182

   ; Set values for xdata and fdata.

xdata   =   1 + 0.1*FINDGEN(91)

fdata   =   f(xdata)

   ; Contaminate the data.

fdata(isub)   =   fdata(isub) + rnoise

   ; Smooth the data.

sdata   = SMOOTHDATA1D(xdata, fdata, Itmax = itmax, $
   Distance = dis, Sc = sc)

   ; Output the results.

PM, [[f(xdata(isub))], [fdata(isub)], [sdata(isub)]], $
       Title   =   '          F(X)     F(X) + noise      sdata'
          F(X)     F(X) + noise      sdata
       9.82958        12.3296       9.87030
       8.26338        5.26338       8.21537
       5.20083        3.20083       5.16823
       2.22328        4.72328       2.26399
       1.25874        4.25874       1.30825
       3.16738        1.16738       3.13830
       7.16751        4.66751       7.13076
       10.8799        12.8799       10.9092
       12.7739        10.7739       12.7075
       7.59407        10.5941       7.63885

   ; Example 2: Specific information available.

dis   =   1.0

sc   =   0.0
```

```
itmax   =   10.0
```

; A warning message is produce because the maximum number
; of iterations is reached.

```
sdata   =   SMOOTHDATA1D(xdata, fdata, Itmax = itmax, $
                            Distance = dis, Sc = sc)
% SMOOTHDATA1D: Warning: MATH_ITMAX_EXCEEDED
```

; Maximum number of iterations limit "ITMAX" = 10 exceeded. The best
; answer found is returned.

; Output the results.

```
PM, [[f(xdata(isub))], [fdata(isub)], [sdata(isub)]], $
       Title   =   '          F(X)     F(X) + noise      sdata'
          F(X)     F(X) + noise       sdata
        9.82958        12.3296       9.83127
        8.26338        5.26338       8.26223
        5.20083        3.20083       5.19946
        2.22328        4.72328       2.22495
        1.25874        4.25874       1.26142
        3.16738        1.16738       3.16958
        7.16751        4.66751       7.16986
        10.8799        12.8799       10.8779
        12.7739        10.7739       12.7699
        7.59407        10.5941       7.59194
```

SCAT2DINTERP Function

Computes a smooth bivariate interpolant to scattered data that is locally a quintic polynomial in two variables.

Usage

result = SCAT2DINTERP(*xydata*, *fdata*, *xout*, *yout*)

Input Parameters

xydata — Two-dimensional array containing the data points for the interpolation problem. Argument *xydata* is dimensioned (2, N_ELEMENTS (*fdata*)). The *i*-th data point (x_i, y_i) is stored in *xydata* (0, *i*) = x_i and *xydata* (1, *i*) = y_i.

fdata — One-dimensional array containing the values to be interpolated.

xout — One-dimensional array specifying the *x* values for the output grid. It must be strictly increasing.

yout — One-dimensional array specifying the *y* values for the output grid. It must be strictly increasing.

Returned Value

result — A two-dimensional array containing the grid of values of the interpolant.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function SCAT2DINTERP computes a C^1 interpolant to scattered data in the plane. Given the data points (in R^3):

$$\{(x_i, y_i, f_i)\}_{i=0}^{n-1}$$

where *n* = N_ELEMENTS(*xydata*) / 2, SCAT2DINTERP returns the values of the interpolant *s* on the user-specified grid. The computation of *s* is as follows:

First, the Delaunay triangulation of the points

$$\{(x_i, y_i)\}_{i=0}^{n-1}$$

is computed. On each triangle T in this triangulation, s has the following form:

$$s(x, y) = \sum_{m+n \le 5} c_{mn}^{T} x^m y^n \qquad \forall (x, y) \in T$$

Thus, s is a bivariate quintic polynomial on each triangle of the triangulation. In addition,

$$s(x_i, y_i) = f_i \qquad \text{for } i = 0, \ldots, n-1$$

and s is continuously differentiable across the boundaries of neighboring triangles. These conditions do not exhaust the freedom implied by the above representation. This additional freedom is exploited in an attempt to produce an interpolant that is faithful to the global shape properties implied by the data. For more information on this procedure, refer to the article by Akima (1978). The output grid is specified by the two real vectors, *xout* and *yout*, that represent the first (second) coordinates of the grid.

Example

In this example, SCAT2DINTERP is used to fit a surface to randomly scattered data. The resulting surface and the original data points are then plotted.

```
RANDOMOPT, Set = 12345
ndata = 15
xydata = FLTARR(2, ndata)
xydata(*) = RANDOM(2 * ndata)
fdata = RANDOM(ndata)
x = xydata(0, *)
y = xydata(1, *)
ngrid = 20
xout = FINDGEN(ngrid)/(ngrid - 1)
yout = FINDGEN(ngrid)/(ngrid - 1)
    ; Define the grid used to evaluate the computed surface.
surf   = $
    SCAT2DINTERP(xydata, fdata, xout, yout)
        ; Call SCAT2DINTERP.
```

```
SURFACE, surf, xout, yout, /Save, $
   Ax = 45, Charsize = 1.5
      ; Plot the computed surface.
PLOTS,  x, y, fdata, /T3d, $
   Symsize = 2, Psym = 2
      ; Plot the original data points.
```

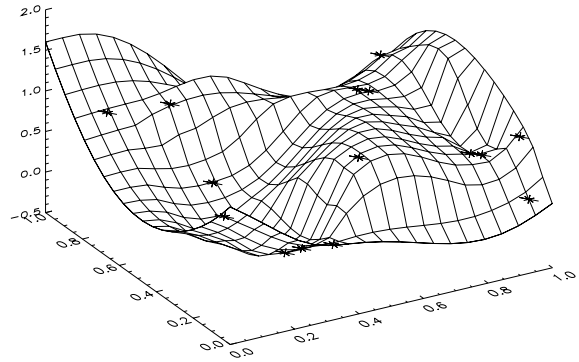


Figure 3-16 Results of fit to scattered data.

Fatal Errors

MATH_DUPLICATE_XYDATA_VALUES — Two-dimensional data values must be distinct.

MATH_XOUT_NOT_STRICTLY_INCRSING — Vector *xout* must be strictly increasing.

MATH_YOUT_NOT_STRICTLY_INCRSING — Vector *yout* must be strictly increasing.

RADBF Function

Computes an approximation to scattered data in R^n for $n \geq 2$ using radial-basis functions.

Usage

result = RADBF(*abscissa*, *fdata*, *num_centers*)

Input Parameters

abscissa — Two-dimensional array containing the abscissas of the data points. Parameter *abscissa* (*i*, *j*) is the abscissa value of the *j*-th data point in the *i*-th dimension.

fdata — One-dimensional array containing the ordinates for the problem.

num_centers — Number of centers to be used when computing the radial-basis fit. Parameter *num_centers* should be less than or equal to N_ELEMENTS (*fdata*).

Returned Value

result — A structure that represents the radial-basis fit.

Input Keywords

Centers — User-supplied centers. See *Discussion* below for details.

Double — If present and nonzero, double precision is used.

Ratio_Centers — Desired ratio of centers placed on an evenly spaced grid to the total number of centers. There is a condition: The same number of centers placed on a grid for each dimension must be equal. Thus, the actual number of centers placed on a grid is usually less than *Ratio_Centers* * *num_centers*, but is never more than *Ratio_Centers* * *num_centers*. The remaining centers are randomly chosen from the set of abscissa given in *abscissa*.

Default: *Ratio_Centers* = 0.5

Random_Seed — Value of the random seed used when determining the random subset of abscissa to use as centers. By changing the value of seed on different calls to RADBF, with the same data set, a different set of random centers are chosen. Setting *Random_Seed* to zero forces the random number seed to

be based on the system clock, so possibly, a different set of centers is chosen each time the program is executed.

Default: *Random_Seed* = 234579

Basis — Character string specifying a user-supplied function to compute the values of the radial functions. The form of the input function is φ (*r*).

Default: the Hardy multiquadratic

Delta — Delta used in the default basis function, ϕ (*r*) = SQRT($r^2 + \delta^2$).

Default: *Delta* = 1

Weights — Requires the user to provide the weights.

Default: all weights equal 1

Discussion

Function RADBF computes a least-squares fit to scattered data in R^d. More precisely, let *n* = N_ELEMENTS (*fdata*), *x* = *abscissa*, *f* = *fdata*, and *d* = N_ELEMENTS (*abscissa* (0, *)). Then,

$$x^0, \ldots, x^{n-1} \in \mathbb{R}^d$$

$$f_0, \ldots, f_{n-1} \in \mathbb{R}^1$$

This function computes a function *F* which approximates the above data in the sense that it minimizes the sum-of-squares error

$$\sum^{n-1} w_i (F(x^i) - f_i)^2$$

where *w* = *Weights*.

The functional form of *F* is, of course, restricted as follows:

$$F(x) := \sum^{k-1} \alpha_j (\sqrt{\|x - c_j\|^2 + \delta^2}) = \sum^{k-1} d_j \phi(\|x - c_j\|)$$

The function ϕ is called the radial function. It maps R^1 into R^1. It needs to be defined only for the nonnegative reals. For the purpose of this routine, the user supplied a function

$$\phi(r) = \sqrt{(r^2 + \delta^2)} .$$

Note that the value of delta is defaulted to 1. It can be set by the user by using keyword *Delta*.

The default-basis function is called the Hardy multiquadric and is defined as

$$\phi(r) = \sqrt{(r^2 + \delta^2)} .$$

A key feature of this routine is the user's control over the selection of the basis function.

In order to obtain the default selection of centers, first compute the number of centers that will be on a grid and the number that will be on a random subset of the abscissa. Next, compute those centers on a grid. Finally, a random subset of abscissa is obtained. This determines where the centers are placed. The selection of centers is discussed in more detail below.

First, the computed grid is restricted to have the same number of grid values in each of the "dimension" directions. Then, the number of centers placed on a grid, *num_gridded*, is computed as follows:

$$\alpha = (Ratio\_Centers)(num\_centers)$$

$$\beta = \lfloor \alpha^{1/dimension} \rfloor$$

$$num\_gridded = \beta^{dimension}$$

Note that there are β grid values in each of the "dimension" directions. Then,

$$num\_random = (num\_centers) - (num\_gridded)$$

How many centers are placed on a grid and how many are placed on a random subset of the abscissa is now known. The gridded centers are computed such that they are equally spaced in each of the "dimension" directions. The last problem is to compute a random subset, without replacement, of the abscissa. The selection is based on a random seed. The default seed is 234579. The user

can change this using optional argument *Random_Seed*. Once the subset is computed, the abscissa as centers is used.

Since the selection of good centers for a specific problem is an unsolved problem at this time, ultimate flexibility is given to the user; that is, the user can select centers using keyword *Centers*. As a rule of thumb, the centers should be interspersed with the abscissa.

The return value for this function is a pointer to the structure containing all the information necessary to evaluate the fit. This pointer is then passed to function RADBE (page 184) to produce values of the fitted function.

Example 1: Fitting Noisy Data using the Default Radial Function

In this example, RADBF is used to fit noisy data. Four plots are generated using different values for *num_centers*. The plots generated by running this example are included after the code. Note that the triangles represent the placement of the centers.

```
PRO radbf_ex1

!P.Multi = [0, 2, 2]
ndata = 10
noise_size = .05
xydata = DBLARR(1, ndata)
fdata = DBLARR(ndata)
    ; Set up parameters.

RANDOMOPT, Set = 234579
    ; Set the random number seed.

noise = 1 - 2 * RANDOM(ndata, /Double)
    ; Generate the noisy data.

xydata(0, *) = 15 * RANDOM(ndata)
fdata = REFORM(COS(xydata(0, *)) + $
    noise_size * noise, ndata)

FOR i = 0, 3 DO BEGIN
    num_centers = ndata/3 + i
        ; Loop on different values of num_centers.
    radial_struct = $
        RADBF(xydata, fdata, num_centers)
        ; Compute the fit.
    a = DBLARR(1, 100)
    a(0, *) = 15 * FINDGEN(100)/99.
```

```
    fit = RADBE(a, radial_struct)
        ; Evaluate fit.

    title = 'Fit with NUM_CENTERS = ' + $
           STRCOMPRESS(num_centers, /Remove_All)

    PLOT, xydata(0, *), fdata, Title = title, $
        Psym = 6, Yrange = [-1.25, 1.25]
        ; Plot results.

    OPLOT, a(0, *), fit
        ; Plot the original data as squares.

    OPLOT, radial_struct.CENTERS, $
        MAKE_ARRAY(num_centers, Value=-1.25), Psym = 5
        ; Plot the x-values of the centers as triangles.
END
END
```

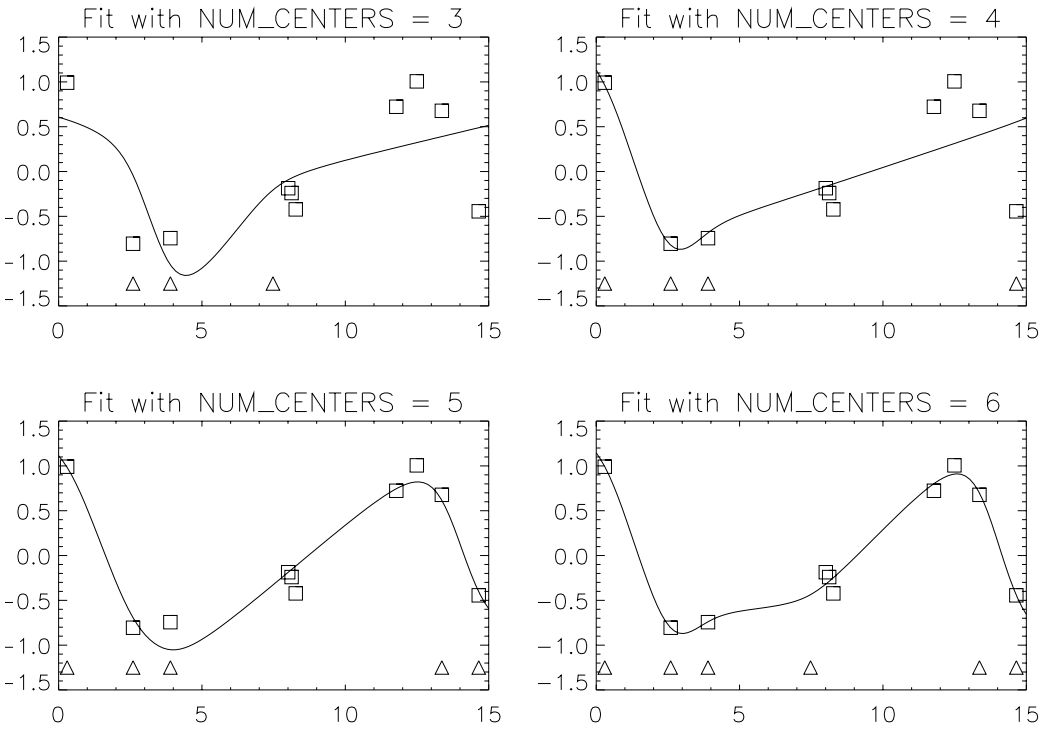


Figure 3-17 Fits using differential values for num_centers.

Example 2: Fitting Noisy Data with a User-supplied Radial Function

This example fits the same data as the first example, but the user supplies the radial function and sets *Ratio_Centers* to zero. The radial function used in this example is $\phi(r) = ln(1 + r^2)$. Four plots are generated using different values for *num_centers*. The plots generated by running this example are included after the code. Note that the triangles represent the placement of the centers.

```
FUNCTION user_fcn, distance
    ; Define the radial function.
RETURN, ALOG(1 + distance^2)
END

PRO radbf_ex2
    ; Set up parameters.
!P.Multi = [0, 2, 2]
ndata = 10
noise_size = .05
xydata = DBLARR(1, ndata)
fdata = DBLARR(ndata)

RANDOMOPT, Set = 234579
    ; Set the random number seed.
noise = 1 - 2 * RANDOM(ndata, /Double)
    ; Generate the noisy data.
xydata(0, *) = 15 * RANDOM(ndata)
fdata = REFORM(COS(xydata(0,*)) + noise_size * noise, ndata)

FOR i = 0, 3 DO BEGIN
    ; Loop on different values of num_centers.
    num_centers = ndata/3 + i

    radial_struct = RADBF(xydata, fdata, $
        num_centers, Ratio_Centers = 0, $
        Basis = 'user_fcn')
        ; Compute the fit.

    a = DBLARR(1, 100)
    a(0, *) = 15 * FINDGEN(100)/99.

    fit = RADBE(a, radial_struct)
        ; Evaluate fit.

    title = 'Fit with NUM_CENTERS = ' + $
        STRCOMPRESS(num_centers, /Remove_All)
```

```
    PLOT, xydata(0,*), fdata, Title = title, $
        Psym = 6, Yrange = [-1.25, 1.25]
        ; Plot results.

    OPLOT, a(0, *), fit
    OPLOT, radial_struct.CENTERS, $
        MAKE_ARRAY(num_centers,Value = -1.25),$
        Psym = 5

END
END
```

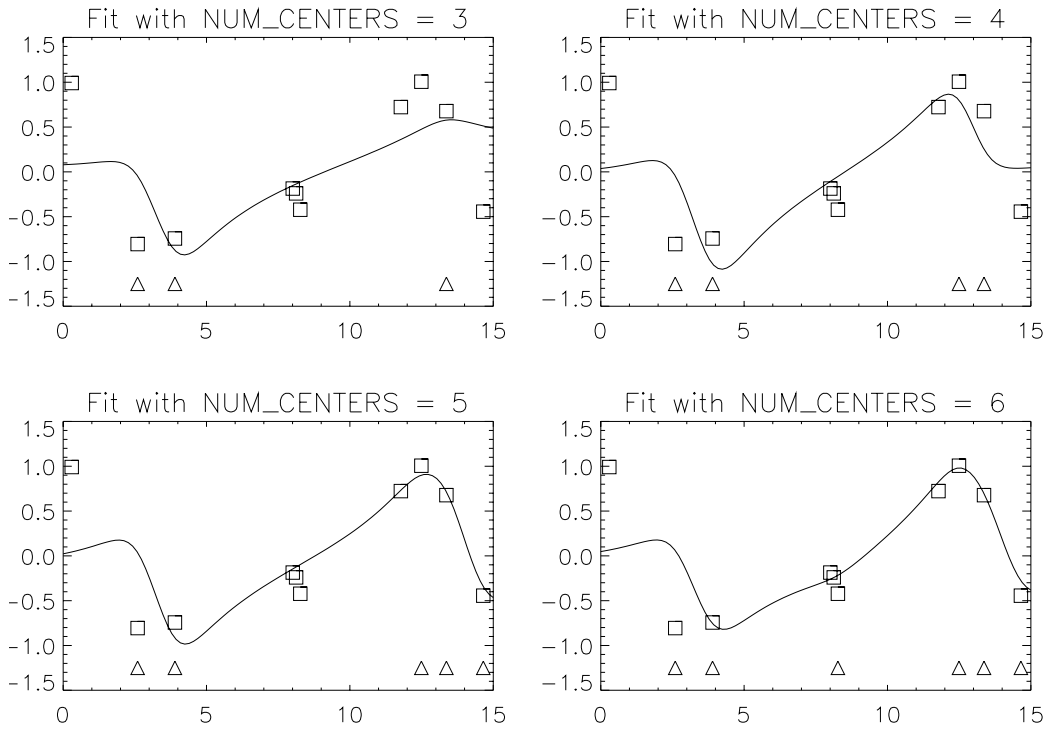


Figure 3-18 Fit using a user-defined radial function.

Example 3: Fitting a Surface to Three-dimensional Scattered Data

This example fits a surface to scattered data. The scattered data is generated using the function $f(x, y) = exp(\ln(y + 1) \sin(x))$. The plots generated by running this example are included after the code.

```
FUNCTION f, x1, x2
    ; This function is used to generate the scattered data function values.
RETURN, EXP(ALOG10(x2 + 1)) * SIN(x1)
END

PRO radbf_ex3
    ; Set up initial parameters.
RANDOMOPT, Set = 123457
ndata            = 50
num_centers     = ndata
xydata           = DBLARR(2, ndata)
fdata            = DBLARR(ndata)
xrange           = 8
yrange           = 5
xydata(0,*) = xrange * RANDOM(ndata, /Double)
xydata(1,*) = yrange * RANDOM(ndata, /Double)
fdata(*)    = f(xydata(0, *), xydata(1, *))
    ; Generate data.
radial_struct = $
    RADBF(xydata, fdata, num_centers, Ratio=0)
        ; Compute fit using RADBF.
WINDOW, /Free
    ; Plot results.
nx = 25
ny = 25
    ; Variables nx and ny are the coarseness of the plotted surfaces.
xyfit        = DBLARR(2, nx * ny)
xyfit(0, *) = xrange * $
    (FINDGEN(nx * ny)/ny)/(nx - 1)
xyfit(1, *) = yrange * $
    (FINDGEN(nx * ny) MOD ny)/(ny - 1)
```

```
zfit = TRANSPOSE(REFORM(RADBE(xyfit, $
   Radial_Struct), ny, nx))
      ; Use TRANSPOSE and REFORM in order to get the results
      ; into a form that SURFACE can use.
xt = xrange * FINDGEN(nx)/(nx-1)
yt = yrange * FINDGEN(ny)/(ny-1)
SURFACE, zfit, xt, yt, /Save, $
   Zrange = [MIN(zfit), MAX(zfit)]
PLOTS, xydata(0, *), xydata(1, *), fdata, $
   /T3d, Psym = 4, Symsize = 2
      ; Plot the original data points over the surface plot.
WINDOW, /Free
orig = DBLARR(nx, ny)
FOR i = 0, (nx-1) DO FOR j = 0, (ny-1) DO $
   orig(i, j) = f(xt(i), yt(j))
SURFACE, orig, xt, yt, $
   Zrange = [MIN(zfit), MAX(zfit)]
      ; Plot original function.
END
```

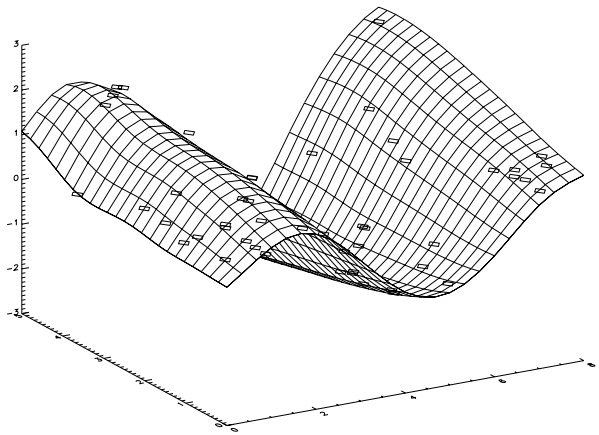


Figure 3-19 Surface fit to scattered data.

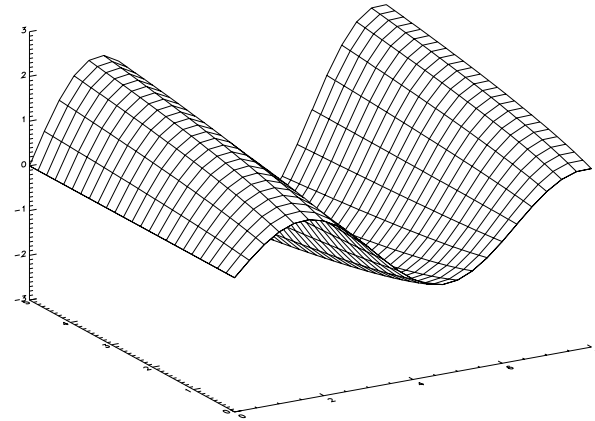


Figure 3-20 Function used to generate scattered data.

RADBE Function

Evaluates a radial-basis fit computed by RADBF.

Usage

result = RADBE(*abscissa*, *radial_fit*)

Input Parameters

abscissa — Two-dimensional array containing the abscissa of the data points at which the fit is evaluated. Argument *abscissa* (*i*, *j*) is the abscissa value of the *j*-th data point in the *i*-th dimension.

radial_fit — Radial-basis structure to be used for the evaluation.

Returned Value

result — An array containing the values of the radial-basis fit at the desired values.

Discussion

Function RADBE evaluates a radial-basis fit from data generated by RADBF. See the documentation for RADBF for details.

Example

See the description of RADBF (page 174) for examples using RADBE.

INTERPOL Function

Standard Library function that linearly interpolates vectors on a regular or irregular grid.

Usage

result = INTERPOL(*v* [, *n*, *x*, *u*])

Input Parameters

v — One-dimensional vector of any type except string.

n — Number of points of result. Both input and output grids are regular. Output grid abscissa value = FLOAT(*i*) / N_ELEMENTS(*v*), for *i* = 0 to *n* – 1.

x — Abscissa values for *v*. This argument is for irregular grids and must have the same number of elements as *v*. Argument *x must* be monotonic, either ascending or descending.

u — Abscissa values for result. This argument is for irregular grids. The result has the same number of elements as *u*, and *u* need not be monotonic.

Returned Value

result — A floating vector of *n* points determined from linearly interpolating input vector. If *v* is double or complex, result is double or complex.

Discussion

The *i*-th element of the vector returned is

$$v(x) + (x - \mathrm{FIX}(x))\ (v(x + 1) - v(x))$$

where

$$x = [i(m - 1)] / (n - 1)$$

for *i* = 0 to *n* – 1; *m* = number of elements in *v* for regular grids. For irregular grids, *x* = *u*(*i*), and *m* = number of points of input vector.

BILINEAR Function

Standard Library function that allows users to determine bilinear interpolate at a set of reference points.

Usage

result = BILINEAR(*p*, *ix*, *jy*)

Input Parameters

p — Two-dimensional data array. Parameters *ix* and *jy* contain the "virtual subscripts" of *p* to look up values for the output.

ix — One- or two-dimensional array. If one-dimensional, *ix* contains the subscripts to look up in *p*. The same set of subscripts is used for all rows in the output array. If two-dimensional, *ix* also contains the "*x*-axis" subscripts except if specified at all points in the output array.

In either case, *ix* must satisfy $0 \leq \min(ix) < n0$ and $0 < \max(ix) \leq n0$, where *n0* is the total number of subscripts in the first dimension of *p*.

jy — One- or two-dimensional array. If *jy* is one-dimensional, it contains the "*y*-axis" subscripts to look up in *p*. The same mask is used in all columns in the output.

In either case, *jy* must satisfy $0 \leq \min(jy) < m0$ and $0 < \max(jy) \leq m0$, where *m0* is the total number of subscripts in the second dimension of *p*.

Returned Value

result — The interpolated array.

Discussion

Function BILINEAR invokes the bilinear interpolation algorithm to evaluate each element in *z* at virtual coordinates contained in *ix* and *jy* with the data in *p*.

Use two-dimensional arrays for *ix* and *jy* when calling BILINEAR because the algorithm is somewhat faster. If *ix* and *jy* are one-dimensional, they are converted to two-dimensional arrays on return from the function. They can then be reused on subsequent calls to save time. The two-dimensional array *p* is unchanged on return.

Example

```
p = FLTARR(3, 3)
ix = [0.1, 0.2]
jy = [0.6, 2.1]
PRINT, BILINEAR(p, ix, jy)
      0.00000       0.00000
      0.00000       0.00000
```

Then, $z(0, 0)$ is returned as though it were equal to $p(0.1, 0.6)$, interpolated from the nearest neighbors at $p(0, 0)$, $p(1, 0)$, $p(1, 1)$, and $p(0, 1)$.

4

CHAPTER

Quadrature

Contents of Chapter

Univariate and Bivariate Quadrature

The first function in this chapter, INTFCN, is designed to compute approximations to integrals of the following form:

$$\int_a^b f(x)w(x)dx$$

or

$$\int_a^b \int_{g(x)}^{h(x)} f(x, y)dxdy$$

The weight function w is used to incorporate known singularities (either algebraic or logarithmic) or to incorporate oscillations. The default action of this function assumes univariate quadrature, a weight function $w(x) = 1$, and the existence of endpoint singularities. Even if no endpoint singularities exist, the default method is still effective for general-purpose integration. If more efficiency is desired, then a more specialized method can be specified through the use of specific parameter and keyword combinations. The available methods can be summarized as follows, where the description refers to subsections of the documentation for the function INTFCN:

- $w(x) = 1$

  Integration of a function with endpoint singularities (default method)

  Integration of a function based on Gauss-Kronrod rules

  Integration of a function with singular points given

  Integration of a function over an infinite or semi-infinite interval

  Integration of a smooth function using a nonadaptive method

  Integration of a two-dimensional iterated integral

- $w(x) = \sin\omega x$ or $w(x) = \cos\omega x$

  Integration of a function containing a sine or cosine factor

  Computing the Fourier sine or cosine transform

- $w(x) = (x - a)^{\alpha} (b - x)^{\beta} \ln(x - a) \ln(b - x)$, where the ln factors are optional

  Integration of functions with algebraic-logarithmic singularities

- $w(x) = 1 / (x - c)$

Integrals in the Cauchy principle value sense

Function INTFCN returns an estimated answer *R* and provides keywords to specify a requested absolute error ε, the requested relative error ρ, and a named variable in which to return an estimate of the error *E*. These numbers are related in the equation below.

$$\left|\int_a^b f(x)w(x)dx - R\right| \le E \le \max\left\{\varepsilon, \rho\left|\int_a^b f(x,y)dydx\right|\right\}$$

One situation that occasionally arises in univariate quadrature concerns the approximation of integrals when only tabular data are given. The functions described above do not directly address this question. However, the standard method for handling this problem is to first interpolate the data, then integrate the interpolant. This can be accomplished by using the PV-WAVE:IMSL Mathematics spline interpolation functions with the spline integration function SPINTEG (page 137), which can be found in Chapter 3, *Interpolation and Approximation*.

Multivariate Quadrature

Two functions, INTFCN and INTFCNHYPER, have been included in this chapter that can be used to approximate certain multivariate integrals.

Function INTFCN can be called with additional parameters and keywords to return an approximation to a two-dimensional iterated integral of the form below.

$$\int_a^b \int_{g(x)}^{h(x)} f(x,y)dydx$$

Function INTFCNHYPER returns an approximation to the integral of a function of *n* variables over a hyper-rectangle as shown in the equation below.

$$\int_{a_0}^{b_0} \cdots \int_{a_{n-1}}^{b_{n-1}} f(x_0, \ldots, x_{n-1})dx_{n-1} \cdots dx_0$$

When working with two-dimensional, tensor-product tabular data, use the PV-WAVE:IMSL Mathematics spline interpolation function BSINTERP, followed by the spline integration function SPINTEG.

Gauss Quadrature

For a fixed number of nodes, N, the Gauss quadrature rule is the unique rule that integrates polynomials of degree less than $2N$. These quadrature rules can be easily computed using procedure GQUAD, which produces the points $\{x_i\}$ and weights $\{w_i\}$ for $i = 1, ..., N$ that satisfy

$$\int_a^b f(x)w(x)dx = \sum_{i=1}^{N} f(x_i)w_i$$

for all functions f that are polynomials of degree less than $2N$. The weight functions w can be selected from the following table:

$w(x)$	Interval	Name
1	(–1, 1)	Legendre
$1/(\sqrt{1-x_2})$	(–1, 1)	Chebyshev 1st kind
$\sqrt{1-x^2}$	(–1, 1)	Chebyshev 2nd kind
e^{-x^2}	(–infinity, infinity)	Hermite
$(1+x)^{\alpha}(1-x)^{\beta}$	(–1, 1)	Jacobi
$e^{-x}x^a$	(0, infinity)	Generalized Laguerre
$1/\cosh(x)$	(–infinity, infinity)	Hyperbolic cosine

Where permissible, GQUAD also computes Gauss-Radau and Gauss-Lobatto quadrature rules.

INTFCN Function

Integrates a user-supplied function. Using different combinations of keywords and parameters, one of several types of integration can be performed including the following:

- Default method
- Integration of functions based on Gauss-Kronrod rules
- Integration of functions with singular points given
- Integration of functions with algebraic-logarithmic singularities
- Integration of functions over an infinite or semi-infinite interval
- Integration of functions containing a sine or cosine factor
- Computation of Fourier sine and cosine transforms
- Integrals in the Cauchy principle value sense
- Integration of smooth functions using a nonadaptive rule
- Computation of two-dimensional iterated integrals

Default Method

Usage

result = INTFCN(*f*, *a*, *b*)

Input Parameters

In the default case, the following three parameters are required. If another method of integration is desired, a combination of these parameters, along with additional parameters and keywords must be specified. For a description of the additional parameters and keywords to use for specific methods of integration, see the *"Optional Methods of Integration"* section on page 196.

f — Scalar string specifying the name of a user-supplied function to be integrated. The function *f* accepts one scalar parameter and returns a single scalar of the same type.

a — Scalar expression specifying the lower limit of integration.

b — Scalar expression specifying the upper limit of integration.

Returned Value

result — An estimate of the desired integral. If no value can be computed, then NaN (Not a Number) is returned.

Global Keywords

The following seven keywords can be used in any combination with each method of integration except the nonadaptive method, which is triggered by the keyword *Smooth*. Because of this, these global keywords are documented here only and referred to within the *Method Keywords* subsections of the *Optional Method of Integrations* section of this routine.

Input

Double — If present and nonzero, double precision is used.

Err_Abs — Absolute accuracy desired.

Default: *Err_Abs* = SQRT(ε), where ε is the machine precision

Err_Rel — Relative accuracy desired.

Default: *Err_Rel* = SQRT(ε), where ε is the machine precision

Max_Subinter — Number of subintervals allowed.

Default: *Max_Subinter* = 500

Output

Err_Est — Named variable in which an estimate of the absolute value of the error is stored.

N_Subinter — Named variable into which the number of subintervals generated is stored.

N_Evals — Named variable into which the number of evaluations of f is stored.

Discussion of Default Method

The default method used by INTFCN is a general-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It subdivides the interval [*a, b*] and uses a 21-point Gauss-Kronrod rule to estimate the integral over each subinterval. The error for each subinterval is estimated by comparison with the 10-point Gauss quadrature rule. The subinterval with the largest estimated error is then bisected, and the same procedure is applied to both halves. The bisection

process is continued until either the error criterion is satisfied, the roundoff error is detected, the subintervals become too small, or the maximum number of subintervals allowed is reached. This method uses an extrapolation procedure known as the ε-algorithm. This method is based on the subroutine QAGS by Piessens et al. (1983).

Should the default method fail to produce acceptable results, consider one of the more specialized methods available by using *method-specific keywords* for this function.

Example

An estimate of

$$\int_0^3 x^2 dx$$

is computed, then compared to the actual value.

```
.RUN
    ; Define the function to be integrated.
- FUNCTION f, x
- RETURN, x^2
- END
ans = INTFCN("f", 0, 3)
    ; Call INTFCN to compute the integral.
PM, 'Computed Answer:', ans
    ; Output the results.
Computed Answer:
     9.00000
PM, 'Exact - Computed:', 3^2 - ans
Exact - Computed:
     0.00000
```

Warning Errors

MATH_ROUNDOFF_CONTAMINATION — Roundoff error, preventing the requested tolerance from being achieved, has been detected.

MATH_PRECISION_DEGRADATION — Degradation in precision has been detected.

MATH_EXTRAPOLATION_ROUNDOFF — Roundoff error in the extrapolation table, preventing the requested tolerance from being achieved, has been detected.

MATH_EXTRAPOLATION_PROBLEMS — Extrapolation table, constructed for convergence acceleration of the series formed by the integral contributions of the cycles, does not converge to the requested accuracy.

MATH_BAD_INTEGRAND_BEHAVIOR — Bad integrand behavior occurred in one or more cycles.

Fatal Errors

MATH_DIVERGENT — Integral is probably divergent or slowly convergent.

MATH_MAX_SUBINTERVALS — Maximum number of subintervals allowed has been reached.

MATH_MAX_CYCLES — Maximum number of cycles allowed has been reached.

MATH_MAX_STEPS — Maximum number of steps allowed have been taken. The integrand is too difficult for this routine.

Optional Methods of Integration

By specifying different sets of parameters and/or keywords, a number of different types of integration can be performed. Internally, the method to be used is determined by examining the combination of parameters and/or keywords used in the call to INTFCN. To specify a specific method of integration, refer to the appropriate discussion.

Integration of Functions Based on Gauss-Kronrod Rules

This method integrates functions using a globally adaptive scheme based on Gauss-Kronrod rules.

Usage

Triggered by the use of keyword *Rule*.

result = INTFCN(*f*, *a*, *b*, *Rule* = *rule*)

Returned Value

result — The value of

$$\int_a^b f(x)dx$$

is returned. If no value can be computed, NaN is returned.

Method Input Keywords

In addition to the keywords listed in the *"Global Keywords"* section, the following keyword is available:

Rule — If specified, the integral is computed using a globally adaptive scheme based on Gauss-Kronrod rules.

Rule	Gauss-Kronrod Rule
1	7-15 points
2	10-21 points
3	15-31 points
4	20-41 points
5	25-51 points
6	30-61 points

Discussion of Integration of Functions Based on Gauss-Kronrod Rules

This method is a general-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It subdivides the interval [*a*, *b*] and uses a (2*k*+1)-point Gauss-Kronrod rule to estimate the integral over each subinterval. The error for each subinterval is estimated by comparison with the *k*-point Gauss quadrature rule. The subinterval with the largest estimated error is then bisected, and the same procedure is applied to both halves. The bisection process is continued until either the error criterion is satisfied, roundoff error is detected, the subintervals become too small, or the maximum number of subintervals allowed is reached. This method is based on the subroutine QAG by Piessens et al. (1983).

If this method is used, the function should be coded to protect endpoint singularities if they exist.

Example

The value of

$$\int_0^1 \sin(1/x)dx$$

is computed. Since the integrand is oscillatory, *Rule* = 6 is used. The exact value is 0.50406706. The values of the actual and estimated error are machine dependent.

```
.RUN
    ; Define the function to be integrated.
- FUNCTION f, x
- RETURN, SIN(1/x)
- END
ans = INTFCN("f", 0, 1, Rule = 6)
    ; Call INTFCN, with Rule = 6, to compute the integral based on the
    ; specified Gauss-Kronrod rule.
PM, 'Computed Answer:',ans
    ; Output the results.
Computed Answer:
    0.504051
exact = .50406706
PM, 'EXACT - COMPUTED:', exact - ans
Exact - Computed:
    1.62125e-05
```

Error Handling

See *Warning Errors* on page 195 and *Fatal Errors* on page 196.

Integration of Functions with Singular Points Given

This method integrates functions with singularity points given.

Usage

Requires the use of keyword *Sing_Pts*.

result = INTFCN(*f*, *a*, *b*, *Sing_Pts* = *points*)

Returned Value

result — The value of

$$\int_a^b f(x)dx$$

is returned. If no value can be computed, NaN is returned.

Method Input Keywords

In addition to the keywords listed in the *"Global Keywords"* section, the following keyword is available:

Sing_Pts — If present, specifies the abscissas of the singularities. These values should be interior to the interval [*a, b*].

Discussion of Integration of Functions with Singular Points Given

This method is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It subdivides the interval [*a, b*] into *N*+1 user-supplied subintervals, where *N* is the number of singular points, and uses a 21-point Gauss-Kronrod rule to estimate the integral over each subinterval. The error for each subinterval is estimated by comparison with the 10-point Gauss quadrature rule. The subinterval with the largest estimated error is then bisected, and the same procedure is applied to both halves. The bisection process is continued until either the error criterion is satisfied, the roundoff error is detected, the subintervals become too small, or the maximum number of subintervals allowed is reached. This method uses an extrapolation procedure known as the ε-algorithm. This method is based on the subroutine QAGP by Piessens et al. (1983).

Example

The value of

$$\int_0^3 x^3 \ln\left|(x^2-1)(x^2-2)\right| dx = 61\ln 2 + \frac{77}{4}\ln 7 - 27$$

is computed. The values of the actual and estimated error are machine dependent. Note that this subfunction never evaluates the user-supplied function at the user-supplied breakpoints.

```
.RUN
    ; Define the function to be integrated.
- FUNCTION f, x
- RETURN, x^3 * ALOG(ABS((x^2 - 1) * $
-   (x^ 2 - 2)))
- END
```

```
ans = INTFCN("f", 0, 3, $
   Sing_Pts = [1, SQRT(2)], N_Evals = nevals)
      ; Call INTFCN using keyword Sing_Pts to specify the singular
      ; points.
PM, 'Computed Answer:', ans
   ; Output the results.
Computed Answer:
     52.7408
exact = 61 * ALOG(2) + (77/4.) * ALOG(7) - 27
PM, 'Exact - Computed:', exact - ans
Exact - Computed:
-2.67029e-05
PM, 'Number of Function Evaluations:', $
   nevals
Number of Function Evaluations:
        819
```

Error Handling

See *Warning Errors* on page 195 and *Fatal Errors* on page 196.

Integration of Functions with Algebraic-logarithmic Singularities

This method integrates functions with algebraic-logarithmic singularities.

Method Input Parameters

alpha — The strength of the singularity at *a*. Must be greater than −1.

beta — Strength of the singularity at *b*. Must be greater than −1.

Usage

Triggered by the use of the parameters *alpha* and *beta* and one of the keywords below in addition to *f*, *a*, and *b*.

result = INTFCN(*f*, *a*, *b*, *alpha*, *beta*, /*Algebraic*)

result = INTFCN(*f*, *a*, *b*, *alpha*, *beta*, /*Alg_Left_Log*)

result = INTFCN(*f*, *a*, *b*, *alpha*, *beta*, /*Alg_Log*)

result = INTFCN(*f, a, b, alpha, beta, /Alg_Right_Log*)

Returned Value

result — The value of

$$\int_a^b f(x)w(x)dx$$

is returned, where $w(x)$ is defined by one of the keywords below. If no value can be computed, NaN is returned.

Method Input Keywords

In addition to the keywords listed in the *"Global Keywords"* section, the following keywords are available. Exactly one of the following keywords must be specified:

Algebraic — If present and nonzero, uses the weight function $(x-a)^{\alpha}(b-x)^{\beta}$. This is the default weight function for this method of integration.

Alg_Left_Log — If present and nonzero, uses the weight function $(x-a)^{\alpha}(b-x)^{\beta}\log(x-a)$.

Alg_Log — If present and nonzero, uses the weight function $(x-a)^{\alpha}(b-x)^{\beta}\log(x-a)\log(x-b)$.

Alg_Right_Log — If present and nonzero, uses the weight function $(x-a)^{\alpha}(b-x)^{\beta}\log(x-b)$.

Discussion of Integration of Functions with Algebraic-logarithmic Singularities

This method is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It computes integrals whose integrands have the special form *w (x) f (x)*, where *w (x)* is a weight function. A combination of modified Clenshaw-Curtis and Gauss-Kronrod formulas is employed. This method is based on the subroutine QAWS, which is fully documented by Piessens et al. (1983).

If this method is used, the function should be coded to protect endpoint singularities if they exist.

Example

The value of

$$\int_0^1 [\ (1+x)(1-x)\ ]^{1/2} \ln(x)\,dx \;=\; \frac{3\ln(2)-4}{9}$$

is computed.

```
.RUN
    ; Define the function to be integrated.
- FUNCTION f, x
- RETURN, SQRT((1 + x))
- END
ans = $
    INTFCN("f", 0, 1, /Alg_Left_Log, 1.0, .5 )
        ; Call INTFCN with keyword Alg_Left_Log set and values for the
        ; method parameters alpha and beta.
PM, 'Computed Answer:', ans
    ; Output the results.
Computed Answer:
   -0.213395
exact = (3 * ALOG(2) - 4)/9
PM, 'Exact - Computed:', exact - ans
Exact - Computed:
    1.49012e-08
```

Error Handling

See *Warning Errors* on page 195 and *Fatal Errors* on page 196.

Integration of Functions Over an Infinite or Semi-infinite Interval

This method integrates functions over an infinite or semi-infinite interval.

Method Input Parameters

bound — The finite limit of integration. If either of the keywords *Inf_Bound* or *Bound_Inf* are specified, this parameter is required.

Usage

Triggered by the presence of the function *f*, a bound *bound*, and one of the keywords *Inf_Inf*, *Inf_Bound*, or *Bound_Inf*.

result = INTFCN(*f*, /*Inf_Inf*)

result = INTFCN(*f*, *bound*, /*Inf_Bound*)

result = INTFCN(*f*, *bound*, /*Bound_Inf*)

Returned Value

result — The value of

$$\int_a^b f(x)\,dx$$

is returned, where *a* and *b* are appropriate integration limits. If no value can be computed, NaN is returned.

Method Input Keywords

In addition to the keywords listed in the *"Global Keywords"* section, the following keywords are available (exactly one of the following keywords must be specified):

Inf_Inf — If present and nonzero, integrates a function over the range (*–infinity, infinity*⟩ .

Inf_Bound — If present and nonzero, integrates a function over the range (*–infinity , bound*).

Bound_Inf — If present and nonzero, integrates a function over the range (*bound*, *infinity*⟩ .

Discussion of Integration of Functions Over an Infinite or Semi-infinite Interval

This method is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It initially transforms an infinite or semi-infinite interval into the finite interval [0, 1]. It then uses the same strategy that is used when specifying the keyword *Sing_Pts*. This method is based on the subroutine QAGI by Piessens et al. (1983).

If this method is used, the function should be coded to protect endpoint singularities if they exist.

Example

The value of

$$\int_0^\infty \frac{\ln(x)}{1 + (10x)^2} dx = \frac{-\pi \ln(10)}{20}$$

is computed.

```
.RUN
    ; Define the function to be integrated.
- FUNCTION f, x
- RETURN, ALOG(x)/(1 + (10 * x)^2)
- END
ans = INTFCN("f", 0, /Bound_Inf)
    ; Call INTFCN with keyword Bound_Inf set. Notice that only the lower
    ; limit of integration is given.
PM, 'Computed Answer:', ans
    ; Output the results.
Computed Answer:
    -0.361689
exact = -!Pi * ALOG(10)/20
PM, 'Exact - Computed:', exact - ans
Exact - Computed:
     5.96046e-08
```

Error Handling

See *Warning Errors* on page 195 and *Fatal Errors* on page 196.

Integration of Functions Containing a Sine or Cosine Factor

This method integrates functions containing a sine or a cosine factor.

Method Input Parameters

omega — The frequency of the trigonometric weighting function.

Usage

Triggered by the use of parameter *omega* and one of the keywords *Sine* or *Cosine* in addition to *f*, *a*, and *b*.

result = INTFCN(*f*, *a*, *b*, *omega*, /*Sine*)

result = INTFCN(*f*, *a*, *b*, *omega*, /*Cosine*)

Returned Value

result — The value of

$$\int_a^b f(x)w(\omega x)dx \ ,$$

where the weight function $w(\omega x)$ is defined by the keywords below, is returned. If no value can be computed, NaN is returned.

Method Input Keywords

In addition to the keywords listed in the *"Global Keywords"* section, the following keywords are available (exactly one of the following keywords must be specified):

Sine — If present and nonzero, $\sin(\omega x)$ is used for the integration weight function.

Cosine — If present and nonzero, $\cos(\omega x)$ is used for the integration weight function.

Max_Moments — A scalar expression specifying an upper bound on the number of Chebyshev moments that can be stored. Increasing (decreasing) this number may increase (decrease) execution speed and space used.

Default: *Max_Moments* = 21

Discussion of Integration of Functions Containing a Sine or Cosine Factor

This method is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It computes integrals whose integrands have the special form $w(x) f(x)$, where $w(x)$ is either $\cos(\omega x)$ or $\sin(\omega x)$. Depending on the length of the subinterval in relation to the size of ω, either a modified Clenshaw-Curtis procedure or a Gauss-Kronrod 7/15 rule is employed to approximate the integral on a subinterval. This method is based on the subroutine QAWO by Piessens et al. (1983).

If this method is used, the function should be coded to protect endpoint singularities if they exist.

Example

The value of

$$\int_0^1 x^2 \sin(3\pi x)\,dx$$

is computed. The following is the exact answer:

$$\frac{(3\pi)^2 - 4}{(3\pi)^3}$$

```
.RUN
    ; Define the function to be integrated.
- FUNCTION f, x
- RETURN, x^2
- END
ans = INTFCN("f", 0, 1, 3 * !Pi, /Sine)
    ; Call INTFCN with keyword Sine set and a value for the method
    ; parameter omega.
PM, 'Computed Answer:', ans
    ; Output the results.

Computed Answer:
     0.101325
exact = ((3 * !Pi)^2 - 2)/((3 * !pi)^3) $
   - 2/(3 * !Pi)^3
PM, 'Exact - Computed:', exact - ans
```

```
Exact - Computed:
    0.00000
```

Error Handling

See *Warning Errors* on page 195 and *Fatal Errors* on page 196.

Computation of Fourier Sine or Cosine Transforms

This method computes Fourier sine or cosine transforms.

Method Input Parameters

omega — The frequency of the trigonometric weighting function.

Usage

Triggered by the use of parameter *omega* and one of the keywords *Sine* or *Cosine* in addition to *f* and *a*.

result = INTFCN(*f*, *a*, *omega*, /*Sine*)

result = INTFCN(*f*, *a*, *omega*, /*Cosine*)

Returned Value

result — The value of

$$\int_a^{\infty} f(x)w(\omega x)dx \ ,$$

where the weight function $w(\omega x)$ is defined by the keywords below, is returned. If no value can be computed, NaN is returned.

Method Input Keywords

In addition to the keywords listed in the *"Global Keywords"* section, the following keywords are available (exactly one of the keywords *Sine* or *Cosine* must be specified):

Sine — If present and nonzero, $\sin(\omega x)$ is used for the integration weight function.

Cosine — If present and nonzero, $\cos(\omega x)$ is used for the integration weight function.

Max_Moments — Number of subintervals allowed in the partition of each cycle.

Default: *Max_Moments* = 21

Method Output Keywords

N_Cycles — Named variable into which the number of cycles generated is stored.

Discussion of Computation of Fourier Sine or Cosine Transforms

This method is a special-purpose integrator that uses a globally adaptive scheme to reduce the absolute error. It computes integrals whose integrands have the special form *w (x) f (x)*, where *w (x)* is either cos (ω*x*) or sin (ω*x*). The integration interval is always semi-infinite of the form [*a*, *infinity*]. This method is based on the subroutine QAWF by Piessens et al. (1983).

If this method is used, the function should be coded to protect endpoint singularities if they exist.

Example

The value of

$$\int_0^{\infty} \frac{\cos\left(\frac{\pi x}{2}\right)}{\sqrt{x}} dx = 1$$

is computed. Notice that the function is coded to protect for the singularity at zero.

```
.RUN
    ; Define the function to be integrated.
- FUNCTION f, x
- IF (x EQ 0) THEN RETURN, x $
-    ELSE RETURN, 1/SQRT(x)
- END
ans = INTFCN("f", 0, !Pi/2, /Cosine)
    ; Call INTFCN with keyword Cosine set and a value for the method
    ; specific parameter omega.
PM, 'Computed Answer:', ans
    ; Output the results.
Computed Answer:
```

```
    0.101325
exact = 1.0
PM, 'Exact - Computed:', exact - ans
Exact - Computed:
    0.00000
```

Error Handling

See *Warning Errors* on page 195 and *Fatal Errors* on page 196.

Integrals in the Cauchy Principle Value Sense

This method computes integrals of the form

$$\int_a^b \frac{f(x)}{x-c}dx$$

in the Cauchy principal value sense.

Method Input Parameters

c — The singular point must not equal *a* or *b*.

Usage

Triggered by the use of parameter *c* and keyword *Cauchy* in addition to *f*, *a*, and *b*.

result = INTFCN(*f*, *a*, *b*, *c*, /*Cauchy*)

Returned Value

result — The value of

$$\int_a^b \frac{f(x)}{x-c}dx$$

is returned. If no value can be computed, NaN is returned.

Method Input Keywords

In addition to the keywords listed in the *"Global Keywords"* section, the following keyword is available (requires the use of keyword *Cauchy*):

Cauchy — If present and nonzero, computes integrals of the form

$$+\int_a^b \frac{f(x)}{x-c}dx$$

in the Cauchy principal value sense.

Discussion of Integrals in the Cauchy Principle Value Sense

This method uses a globally adaptive scheme in an attempt to reduce the absolute error. It computes integrals whose integrands have the special form *w (x) f (x)*, where *w (x)* = $1/(x - c)$. If *c* lies in the interval of integration, then the integral is interpreted as a Cauchy principal value. A combination of modified Clenshaw-Curtis and Gauss-Kronrod formulas is employed. The method is an implementation of the subroutine QAWC by Piessens et al. (1983).

If this method is used, the function should be coded to protect endpoint singularities if they exist.

Example

The Cauchy principal value of

$$\int_{-1}^{5} \frac{1}{x(5x^3+6)}dx = \frac{\ln(25/631)}{18}$$

is computed.

```
.RUN
    ; Define the function to be integrated.
- FUNCTION f, x
- RETURN, 1/(5 * x^3 + 6)
- END
ans = INTFCN("f", -1, 5, 0, /Cauchy)
    ; Call INTFCN with keyword Cauchy set.
PM, 'Computed Answer:', ans
    ; Output the results.
Computed Answer:
   -0.0899440
exact = ALOG(125/631.)/18
PM, 'Exact - Computed:', exact - ans
Exact - Computed:
    1.49012e-08
```

Error Handling

See *Warning Errors* on page 195 and *Fatal Errors* on page 196.

Integration of Smooth Functions Using a Nonadaptive Rule

This method integrates smooth functions using a nonadaptive rule.

Usage

Triggered by the use of keyword *Smooth* in addition to *f*, *a*, and *b*.

result = INTFCN(*f*, *a*, *b*, /*Smooth*)

Returned Value

result — The value of

$$\int_a^b f(x)\,dx$$

is returned. If no value can be computed, NaN is returned.

Method Input Keywords

Because this method is nonadaptive, there are fewer options with the algorithm. For this method, all keywords described in *Global Keywords* on page 194 do not apply. A complete list of the available keywords is given below. This method requires the use of keyword *Smooth.*

Smooth — If present and nonzero, uses a nonadaptive rule to compute the integral.

Double — If present and nonzero, uses double precision.

Err_Abs — Absolute accuracy desired.

Default: *Err_Abs* = SQRT(ε), where ε is the machine precision

Err_Rel — Relative accuracy desired.

Default: *Err_Rel* = SQRT(ε), where ε is the machine precision

Method Output Keywords

Err_Est — Named variable into which an estimate of the absolute value of the error is stored.

Discussion of Integration of Smooth Functions Using a Nonadaptive Method

This method is designed to integrate smooth functions. It implements a nonadaptive quadrature procedure based on nested Paterson rules of order 10, 21, 43, and 87. These rules are positive quadrature rules with degree of accuracy 19, 31, 64, and 130, respectively. This method applies these rules successively, estimating the error until either the error estimate satisfies the user-supplied constraints or the last rule is applied.

This method is not very robust, but for certain smooth functions, it can be efficient. This method is based on the subroutine QNG by Piessens et al. (1983). If this method is used, the function should be coded to protect endpoint singularities if they exist.

Example

The value of

$$\int_0^2 x e^x dx = e^2 + 1$$

is computed.

```
.RUN
    ; Define the function to integrate.
- FUNCTION f, x
- RETURN, x * EXP(x)
- END
ans = INTFCN("f", 0, 2, /Smooth)
    ; Call INTFCN with keyword Smooth set.
PM, 'Computed Answer:', ans
Computed Answer:
     8.38906
exact = EXP(2) + 1
PM, 'Exact - Computed:', exact - ans
Exact - Computed:
  9.53674e-07
```

Error Handling

See *Warning Errors* on page 195 and *Fatal Errors* on page 196.

Integration of Two-dimensional Iterated Integrals

This method integrates two-dimensional iterated integrals.

Method Input Parameters

f — Scalar string specifying the name of a user-supplied PV-WAVE function to be integrated. Function *f* accepts two scalar parameters and returns a single scalar of the same type.

a — Scalar expression specifying the lower limit of the outer integral.

b — Scalar expression specifying the upper limit of the outer integral.

h — Scalar string specifying the name of a user-supplied PV-WAVE function used to evaluate the lower limit of the inner integral. Function *h* accepts one scalar parameter and returns a single scalar of the same type.

g — Scalar string specifying the name of a user-supplied PV-WAVE function used to evaluate the upper limit of the inner integral. Function *g* accepts one scalar parameter and returns a single scalar of the same type.

Usage

Triggered by the use of the parameters *g* and *h* and keyword *Two_Dimensional* in addition to *f*, *a*, and *b*.

result = INTFCN(*f*, *a*, *b*, *g*, *h*, /*Two_Dimensional*)

Returned Value

result — The value of

$$\int_a^b \int_{g(x)}^{h(x)} f(x, y)dydx$$

is returned. If no value can be computed, NaN is returned.

Method Keywords

In addition to the keywords listed in the *"Global Keywords"* section, the following keyword is available and must be specified for this method:

Two_Dimensional — If present and nonzero, integrates a two-dimensional iterated integral.

Discussion of Integration of Two-dimensional Iterated Integrals

This method approximates the following two-dimensional iterated integral:

$$\int_a^b \int_{g(x)}^{h(x)} f(x, y)\,dy\,dx$$

The lower-numbered rules are used for less smooth integrands, while the higher-order rules are more efficient for smooth (oscillatory) integrands.

If this method is used, the function should be coded to protect endpoint singularities if they exist.

Example

In this example, the value of the integral

$$\int_0^1 \int_x^{2x} \sin(x + y)\,dy\,dx$$

is computed.

```
.RUN
    ; Define the function to be integrated.
- FUNCTION f, x, y
- RETURN, SIN(x + y)
- END
.RUN
    ; Define the function for the lower limit of the inner integral.
- FUNCTION g, x
- RETURN, x
- END
.RUN
    ; Define the function for the upper limit of the inner integral.
- FUNCTION h, x
- RETURN, 2 * x
- END
ans = $
   INTFCN("f",0,1,"g","h",/Two_Dimensional)
       ; Call INTFCN with keyword Two_Dimensional set and the
       ; names of the functions defining the limits of the inner integral.
PM, 'Computed Answer:', ans
Computed Answer:
     0.407609
exact = -SIN(3)/3 + SIN(2)/2
```

```
PM, 'Exact - Computed:', exact - ans
Exact - Computed:
 -5.96046e-08
```

Error Handling

See *Warning Errors* on page 195 and *Fatal Errors* on page 196.

INTFCNHYPER Function

Integrates a function on a hyper-rectangle as follows:

$$\int_{a_0}^{b_0} \ldots \int_{a_{n-1}}^{b_{n-1}} f(x_0, \ldots, x_{n-1}) dx_{n-1} \ldots dx_o$$

Usage

result = INTFCNHYPER(*f*, *a*, *b*)

Input Parameters

f — Scalar string specifying the user-supplied function to be integrated. Function *f* accepts as input an array of data points at which the function is to be evaluated and returns the scalar value of the function.

a — One-dimensional array containing the lower limits of integration.

b — One-dimensional array containing the upper limits of integration.

Returned Value

result — The value of the hyper-rectangle function is returned. If no value can be computed, NaN is returned.

$$\int_{a_0}^{b_0} \ldots \int_{a_{n-1}}^{b_{n-1}} f(x_0, \ldots, x_{n-1}) dx_{n-1} \ldots dx_0$$

is returned. If no value can be computed, NaN is returned.

Input Keywords

Err_Abs — Absolute accuracy desired.

Default: *Err_Abs* = SQRT(ε), where ε is the machine precision

Err_Rel — Relative accuracy desired.

Default: *Err_Rel* = SQRT(ε), where ε is the machine precision

Max_Evals — Number of evaluations allowed.

Default: *Max_Evals* = 1,000,000 for $n \leq 2$ and *Max_Evals* = 256^n for $n > 2$, where n is the number of independent variables of f

Output Keywords

Err_Est — Named variable into which an estimate of the absolute value of the error is stored.

Discussion

Function INTFCNHYPER approximates the following n-dimensional iterated integral:

$$\int_{a_0}^{b_0} \ldots \int_{a_{n-1}}^{b_{n-1}} f(x_0, \ldots, x_{n-1}) dx_{n-1} \ldots dx_0$$

An estimate of the error is returned in the optional keyword *Err_Est*. The approximation is achieved by iterated applications of product Gauss formulas. The integral is first estimated by a two-point, tensor-product formula in each direction. Then, for ($i = 0, \ldots, n-1$), the function calculates a new estimate by doubling the number of points in the i-th direction, but halving the number immediately afterwards if the new estimate does not change appreciably. This process is repeated until either one complete sweep results in no increase in the number of sample points in any dimension, the number of Gauss points in one direction exceeds 256, or the number of function evaluations needed to complete a sweep exceeds *Max_Evals*.

Example

In this example, the integral of

$$e^{-(x_0^2 + x_1^2 + x_2^2)}$$

is computed on an expanding cube. The values of the error estimates are machine dependent. The exact integral over R is $\pi^{3/2}$.

```
.RUN
```

```
    ; Define the function to be integrated.
- FUNCTION f, x
- RETURN, EXP(-TOTAL(x^2))
- END
limit = !Pi^1.5
    ; Compute the exact value of the integral.
PM, '     Limit:', limit
    Limit:      5.56833
FOR i = 1, 6 DO  BEGIN $
   a = [-i/2., -i/2., -i/2.] &$
   b = [i/2., i/2., i/2.] &$
   ans = INTFCNHYPER("f", a, b) &$
   PRINT, 'integral = ', ans, $
       ' limit = ', limit
       ; Compute values of the integral over expanding cubes and
       ; output the results after each call to INTFCNHYPER.
 integral = 0.785213 limit = 5.56833
 integral =  3.33231 limit = 5.56833
 integral =  5.02107 limit = 5.56833
 integral =  5.49055 limit = 5.56833
 integral =  5.56135 limit = 5.56833
 integral =  5.56771 limit = 5.56833
```

Warning Errors

MATH_MAX_EVALS_TOO_LARGE — Keyword *Max_Evals* was set too large.

Fatal Errors

MATH_NOT_CONVERGENT — Maximum number of function evaluations has been reached, and convergence has not been attained.

INTFCN_QMC Function

Integrates a function on a hyper-rectangle using a quasi-Monte Carlo method.

Usage

result = INTFCN_QMC(*f*, *a*, *b*)

Input Parameters

f — Scalar string specifying the user-supplied function to be integrated. Function *f* accepts as input an array of data points at which the function is to be evaluated and returns the scalar value of the function.

a — One-dimensional array containing the lower limits of integration.

b — One-dimensional array containing the upper limits of integration.

Returned Value

The value of

$$\int_{a_0}^{b_0} \cdots \int_{a_{n-1}}^{b_{n-1}} f(x_0, \ldots, x_{n-1}) dx_{n-1} \cdots dx_0$$

is returned. If no value can be computed, then NaN is returned.

Input Keywords

Err_Abs — Absolute accuracy desired.

Default: Err_Abs = 1.e-4.

Err_Rel — Relative accuracy desired.

Default: Err_rel = 1.e-4.

Max_Evals — Number of evaluations allowed.

Default: No limit

Base — The value of *BASE* used to compute the Faure sequence.

Skip — The value of *SKIP* used to compute the Faure sequence.

Double — If present and nonzero, double precision is used.

Output Keywords

Err_est — Named variable into which an estimate of the absolute value of the error is stored.

Discussion

Integration of functions over hypercubes by direct methods, such as INTFCN-HYPER, is practical only for fairly low dimensional hypercubes. This is because the amount of work required increases exponentially as the dimension increases.

An alternative to direct methods is Monte Carlo, in which the integral is evaluated as the value of the function averaged over a sequence of randomly chosen points. Under mild assumptions on the function, this method will converge like $1/n^{1/2}$, where n is the number of points at which the function is evaluated.

It is possible to improve on the performance of Monte Carlo by carefully choosing the points at which the function is to be evaluated. Randomly distributed points tend to be non-uniformly distributed. The alternative to a sequence of random points is a *low-discrepancy* sequence. A low-discrepancy sequence is one that is highly uniform.

This function is based on the low-discrepancy Faure sequence, as computed by FAURE_NEXT_PT.

Example

```
FUNCTION F, x
    S = 0.0
    sign = -1.0
    FOR i = 0, N_ELEMENTS(x)-1 DO BEGIN
        prod = 1.0
        FOR j = 0, i DO BEGIN
            prod = prod*x(j)
        END
        S = S + sign*prod
        sign = -sign
    END
    RETURN, s
END

ndim = 10
a = FLTARR(ndim)
a(*) = 0
```

```
b = FLTARR(ndim)
b(*) = 1
result = intfcn_qmc( 'f', a, b)
PM, result
    -0.333010
```

GQUAD Procedure

Computes a Gauss, Gauss-Radau, or Gauss-Lobatto quadrature rule with various classical weight functions.

Usage

GQUAD, *n*, *weights*, *points*

Input Parameters

n — Number of quadrature points.

Output Parameters

weights — Named variable into which an array of length *n* containing the quadrature weights is stored.

points — Named variable into which an array of length *n* containing quadrature points is stored. The default action of this routine is to produce the Gauss Legendre points and weights.

Input Keywords

Double — If present and nonzero, double precision is used.

Cheby_First — Computes the Gauss points and weights using the weight function

$$1/\sqrt{1-x^2}$$

on the interval (–1, 1).

Cheby_Second — Computes the Gauss points and weights using the weight function

$$\sqrt{1-x^2}$$

on the interval (−1, 1).

Hermite — Computes the Gauss points and weights using the weight function exp $(-x^2)$ on the interval (*–infinity, infinity*).

Cosh — Computes the Gauss points and weights using the weight function 1 / cosh (x) on the interval (*–infinity, infinity*).

Jacobi — Specifies an array of length 2 containing the parameters α and β to be used in the weight function $(1 - x)^{\alpha} (1 + x)^{\beta}$. If this keyword is present, computes the Gauss points and weights using the weight function $(1 - x)^{\alpha} (1 + x)^{\beta}$ on the interval (−1, 1).

Laguerre — Specifies the parameter α to be used in the weight function exp $(-x)\ x^{\alpha}$. If this keyword is present, computes the Gauss points and weights using the weight function exp $(-x)\ x^{\alpha}$ on the interval (0, *infinity*).

Fixed_Points — Specifies an array of fixed points. The length of the array can be a maximum of 2.

There are two distinct actions taken depending on the length of this array. If *Fixed_Points* specifies an array of length 1 (a scalar), the procedure computes the Gauss-Radau points and weights using the specified weight function and the fixed point. This formula integrates polynomials of degree less than $2N–1$ exactly. If *Fixed_Points* specifies an array of length 2, the procedure computes the Gauss-Lobatto points and weights using the specified weight function and the fixed points. This formula integrates polynomials of degree less than $2N–2$ exactly.

Discussion

Procedure GQUAD produces the points and weights for the Gauss, Gauss-Radau, or Gauss-Lobatto quadrature formulas for some of the most popular weights. The default weight is the weight function identically equal to 1 on the interval (−1, 1). In fact, it is slightly more general than this suggests because the extra one or two points that can be specified do not have to lie at the endpoints of the interval. This procedure is a modification of the subroutine GAUSSQUADRULE (Golub and Welsch 1969).

In the default case, the procedure returns points in *x = points* and weights in *w = weights* so that

$$\int_a^b f(x)w(x)dx = \sum_{i=0}^{N-1} f(x_i)w_i$$

for all functions *f* that are polynomials of degree less than 2*N*.

If the keyword *Fixed_Points* is specified, then one or two of the above x_i is equal to the values specified by *Fixed_Points*. In general, the accuracy of the above quadrature formula degrades when *n* increases. The quadrature rule integrates all functions *f* that are polynomials of degree less than $2N - F$, where *F* is the number of fixed points.

Example

The three-point Gauss Legendre quadrature points and weights are computed, then used to approximate the integrals as follows:

$$\int_{-1}^{1} x^i dx \qquad i = 0, \ldots, 6$$

Notice that the integrals are exact for the first six monomials, but the last approximation is in error. In general, the Gauss rules with *k*-points integrate polynomials with degree less than 2*k* exactly.

```
GQUAD, 3, weights, points
    ; Call GQUAD to get the weights and points.

error = FLTARR(7)
    ; Define an array to hold the errors.

FOR i = 0, 6 DO error(i) = $
    (TOTAL(weights*(points^i))- $
    (1-(i MOD 2))*2./(i+1))
        ; Compute the errors for seven monomials.

PM, 'Error:', error
    ; Output the results.

Error:
 -2.38419e-07
  2.68221e-07
 -5.96046e-08
  2.08616e-07
  2.98023e-08
  1.78814e-07
 -0.0457142
```

DERIV Function

Performs numerical differentiation using three-point Lagrangian interpolation.

Usage

result = DERIV([*x*,] *y*)

Input Parameters

y — Variable to be differentiated.

x — Differentiates with respect to variable *x*. This parameter is used for unequal point spacing. If omitted, assumes unit spacing for *y*, i.e., *x(i)=i*.

Returned Value

result — Derivative of *y*.

Discussion

See Hildebrand (1956, p. 82).

FCN_DERIV Function

Computes the first, second, or third derivative of a user-supplied function.

Usage

result = FCN_DERIV(*f*, *x*)

Input Parameters

f — Scalar string specifying a user-supplied function whose derivative at *x* will be computed.

x — Point at which the derivative will be evaluated.

Returned Value

result — An estimate of the first, second or third derivative of *f* at *x*. If no value can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Order — The order of the desired derivative (1, 2 or 3).

Default: *Order* = 1

Stepsize — Beginning value used to compute the size of the interval for approximating the derivative. *Stepsize* must be chosen small enough that *f* is defined and reasonably smooth in the interval (*x* − 4.0∗*Stepsize*, *x* + 4.0*Stepsize*), yet large enough to avoid roundoff problems.

Default: *Stepsize* = 0.01

Tolerance — The relative error desired in the derivative estimate. Convergence is assumed when $(2/3)\,|d_2 - d_1| < Tolerance$, for two successive derivative estimates, d_1 and d_2.

Default: *Tolerance* =

$\sqrt[4]{\varepsilon}$

where ε is machine epsilon.

Discussion

The function FCN_DERIV produces an estimate to the first, second, or third derivative of a function. The estimate originates from first computing a spline interpolant to the input function using values within the interval (*x* – 4.0**Stepsize*, *x* + 4.0**Stepsize*), then differentiating the spline at *x*.

Example 1

This example obtains the approximate first derivative of the function $f(x) = -2\sin(3x/2)$ at the point $x = 2$.

```
FUNCTION fcn, x
    f   =   -2*SIN(1.5*x)
    RETURN,   f
END

deriv1   =   FCN_DERIV("fcn", 2.0)
PRINT, "f'(x)    = ", deriv1
f'(x)    =         2.97008
```

Example 2

This example obtains the approximate first, second, and third derivative of the function $f(x) = -2\sin(3x/2)$ at the point $x = 2$.

```
FUNCTION fcn,   x
    f   =   -2*SIN(1.5*x)
    RETURN,   f
END

deriv1   =   FCN_DERIV("fcn", 2.0, /Double)
deriv2   =   FCN_DERIV("fcn", 2.0, ORDER = 2, /Double)
deriv3   =   FCN_DERIV("fcn", 2.0, ORDER = 3, /Double)
PRINT, "f'(x)    = ", deriv1, ",   error =", $
             ABS(deriv1 + 3.0*COS(1.5*2.0))
f'(x)    =          2.9699775,   error =    1.1094893e-07
PRINT, "f''(x)   = ", deriv2, ",   error =", $
```

```
            ABS(deriv2 - 4.5*SIN(1.5*2.0))
f''(x)   =         0.63504004,   error =    5.1086361e-08
PRINT, "f'''(x) = ", deriv3, ",   error =", $
            ABS(deriv3 - 6.75*COS(1.5*2.0))
f'''(x) =         -6.6824494,   error =    1.1606068e-08
```

5

CHAPTER

Differential Equations

Contents of Chapter

Introduction

Ordinary Differential Equations

An *ordinary differential equation* is an equation involving one or more dependent variables called y_i, one independent variable, t, and derivatives of the y_i with respect to t.

In the *initial value problem* (IVP), the initial or starting values of the dependent variables y_i at a known value $t = t_0$ are given. Values of $y_i(t)$ for $t > t_0$ or $t < t_0$ are required.

The function ODE solves the IVP for ODEs of the form

$$\frac{dy_i}{dt} = y_i' = f_i\left(t, y_{0,\cdots} y_{n-1}\right) \qquad i = 0,\ldots,N-1$$

with $y_i(t = t_0)$ specified. Here, f_i is a user-supplied function that must be evaluated at any set of values $(t, y_0, ..., y_{N-1})$, $i = 0, ..., N - 1$.

The above problem statement is abbreviated by writing it as a *system* of first-order ODEs, $y(t) = [y_0(t), ..., y_{N-1}(t)]^T$, $f(t, y) = [f_0(t, y), ..., f_{N-1}(t, y)]^T$, so that the problem becomes $y' = f(t, y)$ with initial values $y(t_0)$.

The system

$$\frac{dy}{dt} = y' = f(t, y)$$

is said to be stiff if some of the eigenvalues of the Jacobian matrix

$$\{(\partial y_i')/(\partial y_j)\}$$

are large and negative. This is frequently the case for differential equations modeling the behavior of physical systems such as chemical reactions proceeding to equilibrium where subspecies effectively complete their reactions in different epochs. An alternate model concerns discharging capacitors such that different parts of the system have widely varying decay rates (or *time constants*).

Users typically identify stiff systems by the fact that certain numerical differential equation solvers, such as the Runge-Kutta-Verner fifth-order and sixth-order method, are inefficient or they fail completely. Special methods are often required. The most common inefficiency is that a large number of evaluations of $f(t, y)$ and, hence, an excessive amount of computer time are required to satisfy the accuracy and stability requirements of the software. In such cases, keyword *R_K_V* should not be specified when using the function ODE. For more about stiff systems, see Gear (1971, Chapter 11) or Shampine and Gear (1979).

Partial Differential Equations

The routine PDE_MOL solves the IVP problem for systems of the form

$$\frac{\partial u_i}{\partial t} = f_i\left(x, t, u_1, \ldots, u_N, \frac{\partial u_1}{\partial x}, \ldots, \frac{\partial u_N}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ldots, \frac{\partial^2 u_N}{\partial x^2}\right)$$

subject to the boundary conditions

$$\begin{aligned} \alpha_1^{(i)} u_i(a) + \beta_1^{(i)} \frac{\partial u_i}{\partial x}(a) &= \gamma_1(t) \\ \alpha_2^{(i)} u_i(b) + \beta_2^{(i)} \frac{\partial u_i}{\partial x}(b) &= \gamma_2(t) \end{aligned}$$

and subject to the initial conditions

$$u_i(x,\ t = t_0) = g_i(x)$$

for $i = 1, \ldots, N$. Here, f_i, g_i,

$$\alpha_j^{(i)}, \text{ and } \beta_j^{(i)}$$

are user-supplied, $j = 1, 2$.

The routine POISSON2D solves Laplace's, Poisson's, or Helmholtz's equation in two dimensions. This routine uses a fast Poisson method to solve a PDE of the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + cu = f(x, y)$$

over a rectangle, subject to boundary conditions on each of the four sides. The scalar constant c and the function f are user specified.

ODE Function

Solves an initial value problem, which is possibly stiff, using the Adams-Gear methods for ordinary differential equations. Using keywords, the Runge-Kutta-Verner fifth-order and sixth-order method can be used if the problem is known not to be stiff.

Usage

result = ODE(*t*, *y*, *f*)

Input Parameters

t — One-dimensional array containing values of the independent variable. Parameter *t*(0) should contain the initial independent variable value, and the remaining elements of *t* should be filled with values of the independent variable at which a solution is desired.

y — Array containing the initial values of the dependent variables.

f — Scalar string specifying a user-supplied function to evaluate the right-hand side. This function takes two parameters, *t* and *y*, where *t* is the current value of the independent variable and *y* is defined above.

The return value of this function is an array defined by the following equation:

$$f(t,y) = \frac{dy}{dt} = y'$$

Returned Value

result — A two-dimensional array containing the approximate solutions for each specified value of the independent variable. The elements (*i*, *) are the solutions for the *i*-th variable.

Input Keywords

Double — If present and nonzero, double precision is used.

Tolerance — Scalar value used to set the tolerance for error control. An attempt is made to control the norm of the local error such that the global error is proportional to *Tolerance*.

Default: *Tolerance* = 0.001

Hinit — Scalar value used for the initial value for the step size *h*. Steps are applied in the direction of integration.

Default: *Hinit* = 0.001 | *t* (*i* + 1) – *t* (*i*) |

Hmin — Scalar value used as the minimum value for the step size *h*.

Default: *Hmin* = 0.0

Hmax — Scalar value used as the maximum value for the step size *h*. If keyword *R_K_V* is set, *Hmax* = 2.0 is used.

Default: largest machine-representable number

Max_Steps — Integer value used in the maximum number of steps allowed per time step.

Default: *Max_Steps* = 500

Max_Evals — Integer value used in the maximum number of function evaluations allowed per time step.

Default: *Max_Evals* = no enforced limit

Scale — Scalar value used as a measure of the scale of the problem, such as an approximation to the Jacobian along the trajectory.

Default: *Scale* = 1

Norm — Switch determining the error norm. In the following, e_i is the absolute value of the error estimate for y_i.

0 Minimum of the absolute error and the relative error equals the maximum of e_i/max ($|y_i|$, 1) for i = 0, …, N_ELEMENTS (y) – 1

1 Absolute error, equals $\max_i e_i$

2 The error norm is $\max_i(e_i/w_i)$, where w_i = max ($|y_i|$, *Floor*)

Default: *Norm* = 0

Floor — Used with *Norm*. Provides a positive lower bound for the error norm option with value 2.

Default: *Floor* = 1.0

R_K_V — If present and nonzero, uses the Runge-Kutta-Verner fifth-order and sixth-order method.

Adams Gear (Default) Method Only

Jacobian — Scalar string specifying a user-supplied function to evaluate the Jacobian matrix. This function takes three parameters, *x*, *y*, and *yprime*, where *x* and *y* are defined in the description of the user-supplied function *f* of the *Input Parameters* section and *yprime* is the array returned by the user-supplied function *f*. The return value of this function is a two-dimensional array containing the partial derivatives. Each derivative $\partial y'_i / \partial y_j$ is evaluated at the provided (x, y) values and is returned in array location (i, j).

Method — Chooses the class of integration methods:

1 Uses implicit Adams method

2 Uses backward differentiation formula (BDF) methods

Default: *Method* = 2

Max_Ord — Defines the highest order formula of implicit Adams type or BDF type to use.

Default: value 12 for Adams formulas; value 5 for BDF formulas

Miter — Chooses the method for solving the formula equations:

1 Uses function iteration or successive substitution

2 Uses chord or modified Newton method and a user-supplied Jacobian matrix

3 Same as 2 except Jacobian is approximated within the function by divided differences

Default: *Miter* = 3

Output Keywords

N_Step — Named variable into which the array containing the number of steps taken at each value of *t* is stored.

N_Evals — Named variable into which the array containing the number of function evaluations used at each value of *t* is stored.

Adams Gear (Default) Method Only

N_Jevals — Named variable into which the array containing the number of Jacobian function evaluations used at each value of *t* is stored. The values returned are nonzero only if the keyword *Jacobian* is also used.

Discussion

Function ODE finds an approximation to the solution of a system of first-order differential equations of the form

$$\frac{dy}{dt} = y' = f(t, y)$$

with given initial conditions for *y* at the starting value for *t*. The function attempts to keep the global error proportional to a user-specified tolerance. The proportionality depends on the differential equation and the range of integration.

The function returns a two-dimensional array with the (*i*, *j*)-th component containing the *i*-th approximate solution at the *j*-th time step. Thus, the returned matrix has dimension (N_ELEMENTS (*y*), N_ELEMENTS (*t*)). It is important to notice here that the initial values of the problem also are included in this two-dimensional matrix.

The code is based on using backward differentiation formulas not exceeding order five as outlined in Gear (1971) and implemented by Hindmarsh (1974). There is an optional use of the code that employs implicit Adams formulas. This use is intended for nonstiff problems with expensive functions $y' = f(t, y)$.

If keyword *R_K_V* is set, the function ODE uses the Runge-Kutta-Verner fifth-order and sixth-order method and is efficient for nonstiff systems where the evaluations of $f(t, y)$ are not expensive. The code is based on an algorithm designed by Hull et al. (1976) and Jackson et al. (1978) and uses Runge-Kutta formulas of order five and six developed by J.H. Verner.

Example 1

This is a mildly stiff example problem (F2) from the test set of Enright and Pryce (1987).

$$y'_0 = -y_0 - y_0 y_1 + k_0 y_1$$

$$y'_1 = -k_1 y_1 + k_2 (1 - y_1) y_0$$

$y_0(0) = 1$

$y_1(0) = 0$

$k_0 = 294.$

$k_1 = 3.$

$k_2 = 0.01020408$

```
.RUN
```
; Define function f.
```
- FUNCTION f, t, y
- RETURN, [-y(0) - y(0) * y(1) + 294. * y(1), $
- -3.*y(1) + 0.01020408*(1. - y(1)) * y(0)]
- END
yp = ODE([0, 120, 240], [1, 0], 'f')
```
; Call the ODE code with the values of the independent variable at
; which a solution is desired and the initial conditions.
```
PM, yp, Format = '(3f10.6)', $
   Title = '    y(0)      y(120)     y(240)'
```
; Output results.
```
    y(0)      y(120)     y(240)
  1.000000  0.514591   0.392082
  0.000000  0.001749   0.001333
```

Example 2: Runge-Kutta Method

This example solves

$$\frac{dy}{dt} = -y$$

over the interval [0, 1] with the initial condition $y(0) = 1$ using the Runge-Kutta-Verner fifth-order and sixth-order method. The solution is $y(t) = e^{-t}$.

```
.RUN
```
; Define function f.
```
- FUNCTION f, t, y
- RETURN, -y
- END
yp = ODE([0, 1], [1], 'f', /R_K_V)
```
; Call ODE with the keyword R_K_V set.

```
PM, yp, Title = 'Solution'
    ; Output results.
Solution
      1.00000     0.367879
PM, yp(1) - EXP(-1), Title = 'Error'
Error
      0.00000
```

Example 3: Predator-Prey Problem

Consider a predator-prey problem with rabbits and foxes. Let r be the density of rabbits, and let f be the density of foxes. In the absence of any predator-prey interaction, the rabbits would increase at a rate proportional to their number, and the foxes would die of starvation at a rate proportional to their number. Mathematically, the model without species interaction is approximated by the following equations:

$$r' = 2r$$
$$f' = -f$$

With species interaction, the rate at which the rabbits are consumed by the foxes is assumed to equal the value $2rf$. The rate at which the foxes increase because they are consuming the rabbits, is equal to rf. Thus, the model differential equations to be solved are as follows:

$$r' = 2r - 2rf$$
$$f' = -f + rf$$

For illustration, the initial conditions are taken to be $r(0) = 1$ and $f(0) = 3$. The interval of integration is $0 \le t \le 40$. In the program, $y(0) = r$ and $y(1) = f$. Function ODE is then called with 100 time values from 0 to 40.

```
.RUN
    ; Define the function f.
- FUNCTION f, t, y
- yp = y
- yp(0) = 2 * y(0) * (1 - y(1))
- yp(1) = -y(1) * (1 - y(0))
- RETURN, yp
- END
```

```
y = [1, 3]
    ; Set the initial values and time values.
t = 40 * FINDGEN(100)/99
y = ODE(t, y, 'f', /R_K_V)
    ; Call ODE with R_K_V set to use the Runge-Kutta method.
!P.Font = 0
    ; Use hardware font.
PLOT, y(0, *), y(1, *), Psym = 2, $
   XTitle = 'Density of Rabbits', $
   YTitle = 'Density of Foxes'
       ; Plot the result.
```

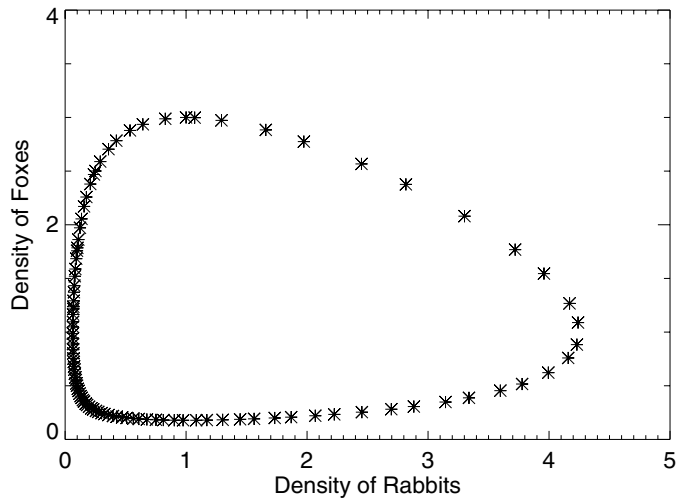


Figure 5-1 Plot of predator-prey example.

Example 4: Stiff Problems and Changing Defaults

This problem is a stiff example (F5) from the test set of Enright and Pryce (1987). An initial step size of $h = 10^{-7}$ is suggested by these authors. When solving a problem that is known to be stiff, using double precision is advisable. Function ODE is forced to use the suggested initial step size and double precision by using keywords.

$$y'_0 = k_0 (-k_1 y_0 y_1 + k_2 y_3 - k_3 y_0 y_2)$$

$$y'_1 = - k_0 k_1 y_0 y_1 + k_4 y_3$$

$$y'_2 = k_0 (-k_3 y_0 y_2 + k_5 y_3)$$

$$y'_3 = k_0 (k_1 y_0 y_1 - k_2 y_3 + k_3 y_0 y_2)$$

$$y_0(0) = 3.365 \times 10^{-7}$$

$$y_1(0) = 8.261 \times 10^{-3}$$

$$y_2(0) = 1.641 \times 10^{-3}$$

$$y_3(0) = 9.380 \times 10^{-6}$$

$$k_0 = 10^{11}$$

$$k_1 = 3.$$

$$k_2 = 0.0012$$

$$k_3 = 9.$$

$$k_4 = 2 \times 10^{7}$$

$$k_5 = 0.001$$

```
.RUN
```

; Define the function.

```
- FUNCTION f, t, y
- k  = [1d11, 3., .0012, 9., 2d7, .001]
- yp = [k(0)*(-k(1)*y(0)*y(1)+k(2)*y(3)- $
- k(3)*y(0)*y(2)),-k(0)*k(1)*y(0)*y(1)+ $
- k(4)*y(3),k(0)*(-k(3)*y(0)*y(2) + $
- k(5)*y(3)),k(0)* (k(1)*y(0)*y(1)- $
- k(2)*y(3)+k(3)*y(0)*y(2))]
- RETURN, yp
- END
```

```
t = FINDGEN(500)/5e6
```

; Set up the values of the independent variable.

```
y = [3.365e-7, 8.261e-3, 1.641e-3, 9.380e-6]
```

; Set the initial values.

```
y = ODE(t, y, 'f', Hinit = 1d-7, /Double)
```

; Call ODE.

```
!P.Multi = [0, 2, 2]
!P.Font = 0
PLOT, t, y(0, *), Title = '!8y!I0!5', XTics=2
PLOT, t, y(1, *), Title = '!8y!I1!5', XTics=2
PLOT, t, y(2, *), Title = '!8y!I2!5', XTics=2
PLOT, t, y(3, *), Title = '!8y!I3!5', XTics=2
```

; Plot each variable on a separate axis.

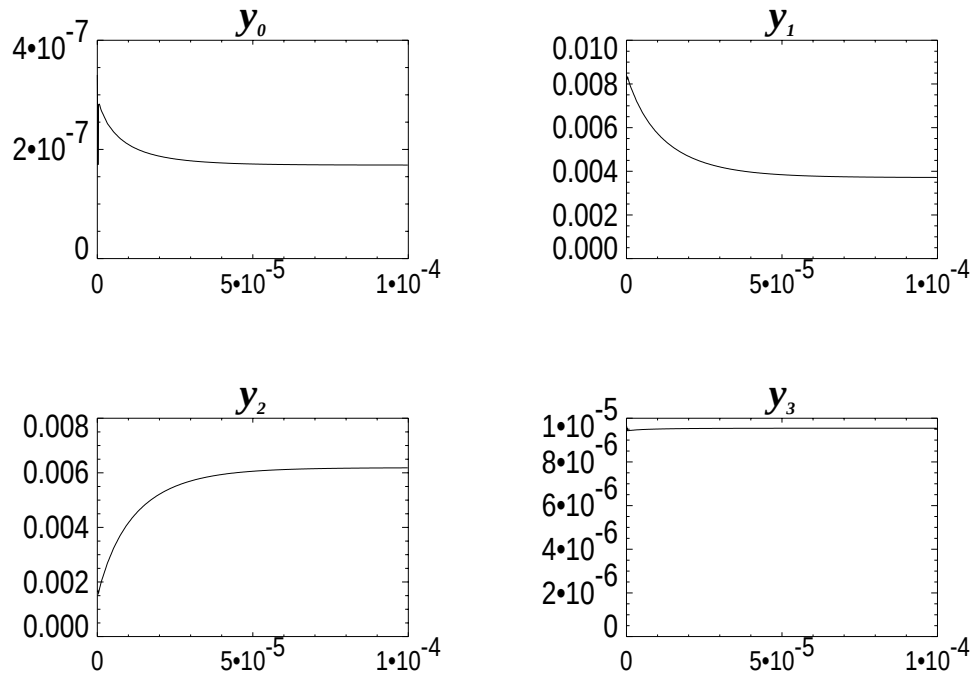


Figure 5-2 Plot for each variable.

Example 5: Strange Attractors—The Rossler System

In this example, a strange attractor is illustrated. The strange attractor used is the Rossler system, a simple model of a truncated Navier-Stokes equation. The Rossler system is given by relation below.

$$y'_0 = -y_1 - y_2$$

$$y'_1 = y_0 + a\, y_1$$

$$y'_2 = b + y_0\, y_2 - c\, y_2$$

The initial conditions and constants are shown below.

$$y_0(0) = 1$$

$$y_1(0) = 0$$

$$y_2(0) = 0$$

$a = 0.2$

$b = 0.2$

$c = 5.7$

```
.RUN
    ; Define function f.
- FUNCTION f, t, y
- COMMON constants, a, b, c
    ; Define some common variables.
- yp = y
- yp(0) = -y(1) - y(2)
- yp(1) = y(0) + a * y(1)
- yp(2) = b + y(0) * y(2) - c * y(2)
- RETURN, yp
- END
COMMON constants, a, b, c
a = .2
b = .2
c = 5.7
    ; Assign values to the common variables.
ntime = 5000
    ; Set the number of values of the independent variable.
time_range = 200
    ; Set the range of the independent variable to 0, ..., 200.
max_steps = 20000
    ; Allow up to 20,000 steps per value of the independent variable.
t = FINDGEN(ntime)/(ntime - 1) * time_range
y = [1, 0, 0]
    ; Set the initial conditions.
y = ODE(t, y, "f", Max_Steps = max_steps, $
   /Double)
       ; Call ODE using keywords Max_Steps and Double.
!P.Charsize = 1.5
SURFACE, FINDGEN(2, 2), /Nodata, $
   XRange = [MIN(y(0, *)), MAX(y(0, *))], $
   YRange = [MIN(y(1, *)), MAX(y(1, *))], $
   ZRange = [MIN(y(2, *)), MAX(y(2, *))], $
```

```
    XTitle = '!6y!i0', YTitle = 'y!i1', $
    ZTitle = 'y!i2', Az = 25, /Save
PLOTS, y(0, *), y(1, *), y(2, *), /T3d
    ; Set up axes to plot the solution. The call to SURFACE draws the
    ; axes and defines the transformation used in PLOTS. The
    ; transformation is saved using keyword Save in SURFACE, then
    ; applied in PLOTS by setting keyword T3d.
```

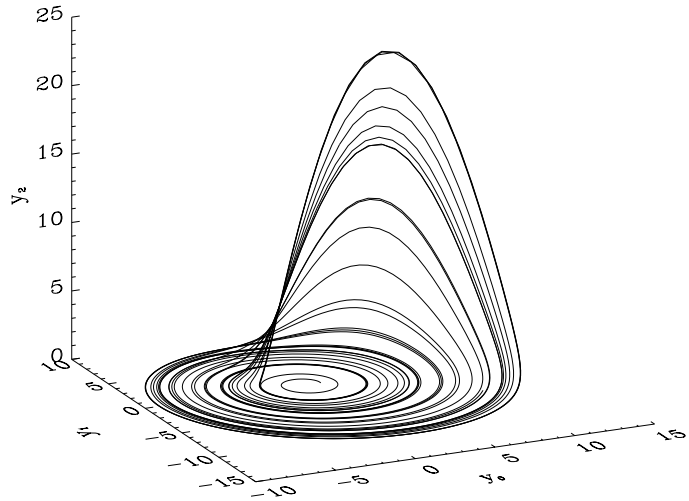


Figure 5-3 Plot of Rossler system.

Example 6: Coupled, Second-order System

Consider the two-degrees-of-freedom system represented by the model (and corresponding free-body diagrams) in *Figure 5-4*. Assuming y_1 is greater than y_0 causes the spring k_1 to be in tension, as seen by the tensile force $k_1 (y_1 - y_0)$.

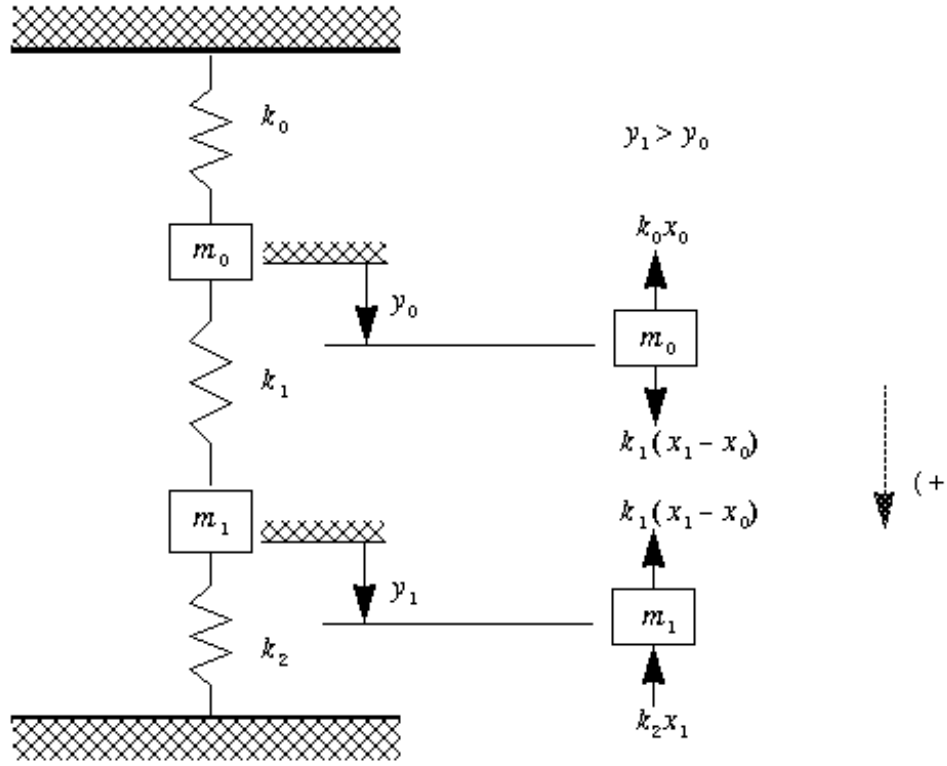


Figure 5-4 Two-degrees-of-freedom system.

NOTE If y_0 is taken to be greater than y_1, then spring k_1 is in compression, with the spring force $k_1 (y_0 - y_1)$. Both methods give correct results when a summation of forces is written.

The differential equations of motion for the system are written as follows:

$$m_0 \ddot{y}_0 = -k_0 y_0 + k_1 (y_1 - y_0)$$

$$m_1 \ddot{y}_1 = -k_1 (y_1 - y_0) - k_2 y_1$$

Thus,

$$\ddot{y}_0 = -\left(\frac{k_0 + k_1}{m_0}\right)y_0 + \left(\frac{k_1}{m_0}\right)y_1$$

$$\ddot{y}_1 = \left(\frac{k_1}{m_1}\right)y_0 - \left(\frac{k_1 + k_2}{m_1}\right)y_1$$

If given the mass and spring constant values

$$m_0 = m_1 = 1\text{kg}$$

$$k_0 = k_1 = k_2 = 1000\frac{N}{m}$$

the following is true:

$$\ddot{y}_0 = (-2000)y_0 + (1000)y_1$$

$$\ddot{y}_1 = (1000)y_0 - 2000y_1$$

Now, in order to convert this problem into one which ODE can be used to solve, choose the following variables:

$$z(0) = y_0$$

$$z(1) = y_1$$

$$z(2) = \dot{y}_0$$

$$z(3) = \dot{y}_1$$

$$k(0) = -2000$$

$$k(1) = 1000$$

which yields the following equations:

$$\dot{y}_0 = z(2)$$

$$\dot{y}_1 = z(3)$$

$$\ddot{y}_0 = k(0)z(0) + k(1)z(1)$$

$$\ddot{y}_1 = k(1)z(0) + k(0)z(1)$$

The last four equations are the object of the return values of the user-supplied function in the exact order as specified above.

The example below loops through four different sets of initial values for *z*.

```
.RUN
    ; Define a function.
- FUNCTION f, t, z
- k = [-2000, 1000]
- RETURN, [z(2), z(3), k(0) * z(0) + k(1) * $
-   z(1), k(1) * z(0) + k(0) * z(1)]
- END
t = FINDGEN(1000)/999
    ; Independent variable, t, is between 0 and 1.
!P.Multi = [0, 2, 2]
    ; Place all four plots in one window.
FOR i = 0, 3 DO BEGIN
z = [1, i/3., 0, 0]
z = ODE(t, z, 'f', Max_Steps = 1000, $
   Hinit = 0.001, /R_K_V)
PLOT, t, z(0, *), Thick = 2, $
   Title = 'Displacement of Mass'
       ; Plot the displacement of m0 as a solid line.
OPLOT, t, z(1, *), Linestyle = 1, Thick = 2
    ; Overplot the displacement of m1 as a dotted line.
END
END
```

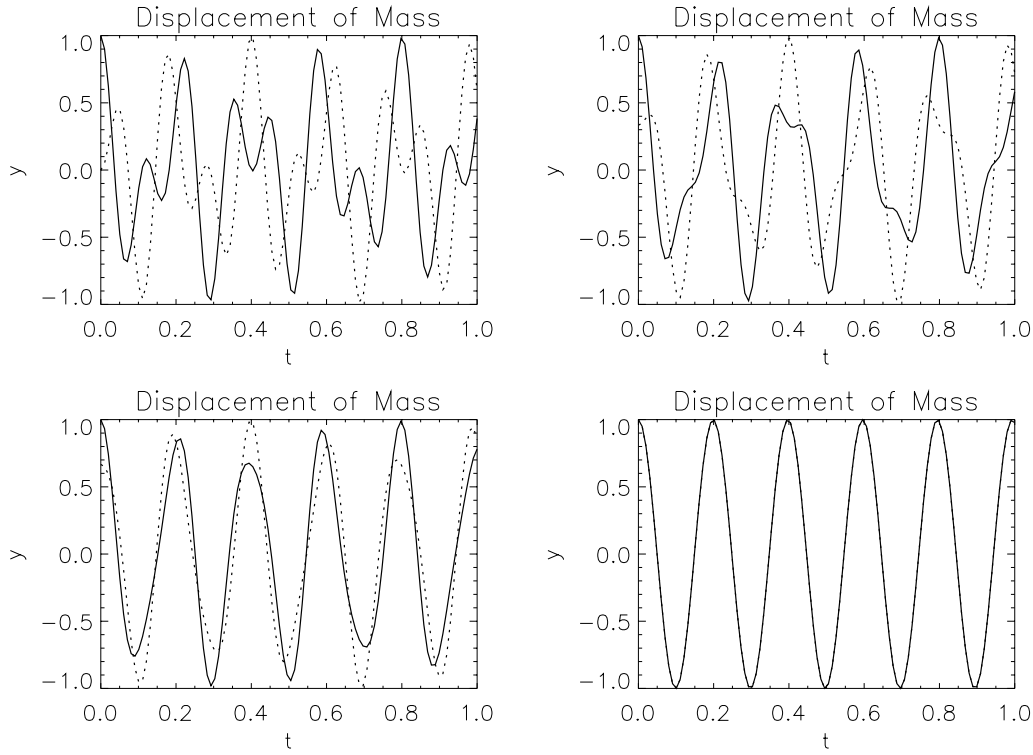


Figure 5-5 Second-order systems with different initial values.

The displacement for m_0 is the solid line, and the dotted line represents the displacement for m_1. Note that when the initial conditions for

$$\dot{y}_0 \text{ and } \dot{y}_1$$

are equal, the displacement of the masses is equal for all values of the independent variable (as seen in the fourth plot). Also, the two principal modes of this problem occur when the following is true:

$$\dot{y}_0 = \dot{y}_1 = 1$$

$$\ddot{y}_0 = 1, \ddot{y}_1 = 1$$

Fatal Errors

MATH_ODE_TOO_MANY_EVALS — Completion of the next step would make the number of function evaluations #, but only # evaluations are allowed.

MATH_ODE_TOO_MANY_STEPS — Maximum number of steps allowed; # used. The problem may be stiff.

MATH_ODE_FAIL — Unable to satisfy the error requirement. *Tolerance* = # may be too small.

PDE_MOL Function

Solves a system of partial differential equations of the form $u_t = f(x, t, u, u_x, u_{xx})$ using the method of lines. The solution is represented with cubic Hermite polynomials.

Usage

result = PDE_MOL(*t, y, xbreak, f_ut, f_bc*)

Input Parameters

t — One-dimensional array containing values of independent variable. Element *t*(0) should contain the initial independent variable value (the initial time, t_0) and the remaining elements of *t* should be values of the independent variable at which a solution is desired.

y —Two-dimensional array of size npde by nx containing the initial values, where *npde* is the number of differential equations and nx is the number of mesh points or lines. It must satisfy the boundary conditions.

xbreak — One-dimensional array of length *nx* containing the breakpoints for the cubic Hermite splines used in the *x* discretization. The points in *xbreak* must be strictly increasing. The values *xbreak*(0) and *xbreak*(nx − 1) are the endpoints of the interval.

f_ut — Scalar string specifying an user-supplied function to evaluate u_t. Function *f_ut* accepts the following input parameters:

> ***npde*** — Number of equations.
>
> ***x*** — Space variable, *x*.

t — Time variable, *t*.

u — One-dimensional array of length *npde* containing the dependent values, *u*.

ux — One-dimensional array of length *npde* containing the first derivatives, u_x.

uxx — One-dimensional array of length *npde* containing the second derivative, u_{xx}.

The return value of this function is an one-dimensional array of length *npde* containing the computed derivatives u_t

f_bc — Scalar string specifying an user-supplied procedure to evaluate the boundary conditions. The boundary conditions accepted by PDE_MOL are

$$\alpha_k u_k + \beta_k \frac{\partial u_k}{\partial x} = \gamma_k$$

NOTE Users must supply the values α_k *and* β_k, which determine the values γ_k. Since γ_k can depend on t values, γ_k' also are required.

---

npde — Number of equations. (Input)

x — Space variable, *x*. (Input)

t — Time variable, *t*. (Input)

alpha — Named variable into which an one-dimensional array of length *npde* containing the α_k values is stored. (Output)

beta — Named variable into which an one-dimensional array of length *npde* containing the β_k values is stored. (Output)

gammap — Named variable into which an one-dimensional array of length *npde* containing the derivatives,

$$\frac{d\gamma_k}{dt} = \gamma_k'$$

is stored. (Output)

Returned Value

result — Three-dimensional array of size *npde* by nx by N_ELEMENTS(*t*) containing the approximate solutions for each specified value of the independent variable.

Input Keywords

Double — If present and nonzero, double precision is used.

Tolerance — Differential equation error tolerance. An attempt is made to control the local error in such a way that the global relative error is proportional to *Tolerance*.

Default: *Tolerance* = 100.0∗ε, ωηερε ε is machine epsilon.

Hinit — Initial step size in the *t* integration. This value must be nonnegative. If *Hinit* is zero, an initial step size of $0.001|t_{i+1} - t_i|$ will be arbitrarily used. The step will be applied in the direction of integration.

Default: *Hinit* = 0.0

Deriv_Init — Two-dimensional array that supplies the derivative values $u_x(x, t(0))$. This derivative information is input as

$$\text{Deriv\_Init}(k,i) = \frac{\partial u_k}{\partial u_x}\left(x, t(0)\right)$$

Default: Derivatives are computed using cubic spline interpolation

Discussion

Let M = *npde*, N = nx and x_i = *xbreak*(*i*). The routine PDE_MOL uses the method of lines to solve the partial differential equation system

$$\frac{\partial u_k}{\partial t} = f_k\left(x, t, u_1, \ldots u_M, \frac{\partial u_1}{\partial x}, \ldots \frac{\partial u_M}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ldots \frac{\partial^2 u_M}{\partial x^2}\right)$$

with the initial conditions

$u_k = u_k(x, t)$ at $t = t_0$, where $t_0 = t(0)$

and the boundary conditions

$$\alpha_k u_k + \beta_k \frac{\partial u_k}{\partial x} = \gamma_k \qquad \text{at } x = x_1 \text{ and at } x = x_N$$

for $k = 1, \ldots, M$.

Cubic Hermite polynomials are used in the x variable approximation so that the trial solution is expanded in the series

$$\hat{u}_k(x,t) = \sum_{i=1}^{N} \left(a_{i,k}(t)\phi_i(x) + b_{i,k}(t)\psi_i(x) \right)$$

where $\phi_i(x)$ and $\psi_i(x)$ are the standard basis functions for the cubic Hermite polynomials with the knots $x_1 < x_2 < \ldots < x_N$. These are piecewise cubic polynomials with continuous first derivatives. At the breakpoints, they satisfy

$$\phi_i(x_l) = \delta_{il} \qquad \psi_i(x_l) = 0$$
$$\frac{d\phi_i}{dx}(x_l) = 0 \qquad \frac{d\psi_i}{dx}(x_l) = \delta_{il}$$

According to the collocation method, the coefficients of the approximation are obtained so that the trial solution satisfies the differential equation at the two Gaussian points in each subinterval,

$$p_{2j-1} = x_j + \frac{3-\sqrt{3}}{6}\left(x_{j+1} - x_j\right)$$
$$p_{2j} = x_j + \frac{3-\sqrt{3}}{6}\left(x_{j+1} + x_j\right)$$

for $j = 1, \ldots, N$. The collocation approximation to the differential equation is

$$\frac{da_{i,k}}{dt}\phi_i(p_j) + \frac{db_{i,k}}{dt}\psi_i(p_j) =$$
$$f_k\left(p_j, t, \hat{u}_1(p_j), \ldots, \hat{u}_M(p_j), \ldots, (\hat{u}_1)_{xx}(p_j), \ldots, (\hat{u}_M)_{xx}(p_j)\right)$$

for $k = 1, \ldots, M$ and $j = 1, \ldots, 2(N - 1)$.

This is a system of $2M(N - 1)$ ordinary differential equations in $2M\ N$ unknown coefficient functions, $a_{i,k}$ *and* $b_{i,k}$. This system can be written in the matrix–vector form as $A\ dc/dt = F\ (t, y)$ with $c(t_0) = c_0$ where c is a vector of coefficients of length $2M\ N$ and c_0 holds the initial values of the coefficients. The last $2M$ equations are obtained by differentiating the boundary conditions

$$\alpha_k \frac{da_k}{dt} + \beta_k \frac{db_k}{dt} = \frac{d\gamma_k}{dt}$$

for $k = 1, \ldots, M$.

The initial conditions $u_k(x, t_0)$ must satisfy the boundary conditions. Also, the $\gamma_k(t)$ must be continuous and have a smooth derivative, or the boundary conditions will not be properly imposed for $t > t_0$.

If $\alpha_k = \beta_k = 0$, it is assumed that no boundary condition is desired for the k-th unknown at the left endpoint. A similar comment holds for the right endpoint. Thus, collocation is done at the endpoint. This is generally a useful feature for systems of first-order partial differential equations.

If the number of partial differential equations is $M = 1$ and the number of break-points is $N = 4$, then

$$A = \begin{bmatrix} \alpha_1 & \beta_1 & & & & & & \\ \phi_1(p_1) & \psi_1(p_1) & \phi_2(p_1) & \psi_2(p_1) & & & & \\ \phi_1(p_2) & \psi_1(p_2) & \phi_2(p_2) & \psi_2(p_2) & & & & \\ & & \phi_3(p_3) & \psi_3(p_3) & \phi_4(p_3) & \psi_4(p_3) & & \\ & & \phi_3(p_4) & \psi_3(p_4) & \phi_4(p_4) & \psi_4(p_4) & & \\ & & & & \phi_5(p_5) & \psi_5(p_5) & \phi_6(p_5) & \psi_6(p_5) \\ & & & & \phi_5(p_6) & \psi_5(p_6) & \phi_6(p_6) & \psi_6(p_6) \\ & & & & & & \alpha_4 & \beta_4 \end{bmatrix}$$

The vector c is

$$c = [a_1, b_1, a_2, b_2, a_3, b_3, a_4, b_4]^T$$

and the right-side F is

$$F = \left[\gamma'(x_1), f(p_1), f(p_2), f(p_3), f(p_4), f(p_5), f(p_6), \gamma'(x_4)\right]^T$$

If $M > 1$, then each entry in the above matrix is replaced by an $M \times M$ diagonal matrix. The element α_1 is replaced by $\text{diag}(\alpha_{1,1}, \ldots, \alpha_{1,M})$. The elements α_N, β_1 and β_N are handled in the same manner. The $\phi_i(p_j)$ *and* $\psi_i(p_j)$ elements are replaced by $\phi_i(p_j)I_M$ *and* $\psi_i(p_j)I_M$ where I_M is the identity matrix of order M. See Madsen and Sincovec (1979) for further details about discretization errors and Jacobian matrix structure.

The input array y contains the values of the $a_{k,i}$. The initial values of the $b_{k,i}$ are obtained by using the PV-WAVE cubic spline routine CSINTERP (Chapter 3: *Interpolation and Approximation)* to construct functions

$$\hat{u}_k(x, t_0)$$

such that

$$\hat{u}_k(x_i, t_0) = a_{ki}$$

The PV-WAVE routine SPVALUE, Chapter 3, “Interpolation and Approximation” is used to approximate the values

$$\frac{d\hat{u}_k}{dx}(x_i, t_0) \equiv b_{k,i}$$

There is an optional use of PDE_MOL that allows the user to provide the initial values of $b_{k,i}$.

The order of matrix A is $2MN$ and its maximum bandwidth is $6M - 1$. The band structure of the Jacobian of F with respect to c is the same as the band structure of A. This system is solved using a modified version of ODE, (page 230). Some of the linear solvers were removed. Numerical Jacobians are used exclusively. The algorithm is unchanged. Gear’s BDF method is used as the default because the system is typically stiff.

Four examples of PDEs are now presented that illustrate how users can interface their problems with PDE_MOL. The examples are small and not indicative of the complexities that most practitioners will face in their applications.

Examples

Example 1

The normalized linear diffusion PDE, $u_t = u_{xx}$, $0 \le x \le 1$, $t > t_0$, is solved. The initial values are $t_0 = 0$, $u(x, t_0) = u_0 = 1$. There is a “zero-flux” boundary condition at $x = 1$, namely $u_x(1, t) = 0$, $(t > t_0)$. The boundary value of $u(0, t)$ is abruptly changed from u_0 to the value $u_1 = 0.1$. This transition is completed by $t = t_\delta = 0.09$.

Due to restrictions in the type of boundary conditions successfully processed by PDE_MOL, it is necessary to provide the derivative boundary value function γ' at $x = 0$ and at $x = 1$. The function γ at $x = 0$ makes a smooth transition from the value u_0 at $t = t_0$ to the value u_1 at $t = t_\delta$. The transition phase for γ' is computed by evaluating a cubic interpolating polynomial. For this purpose, the

function subprogram SPVALUE, Chapter 3: *Interpolation and Approximation* is used. The interpolation is performed as a first step in the user-supplied procedure *f_bc*. The function and derivative values $\gamma(t_0) = u_0$, $\gamma'(t_0) = 0$, $\gamma(t_\delta) = u_1$, and $\gamma'(t_\delta) = 0$, are used as input to routine CSINTERP to obtain the coefficients evaluated by SPVALUE. Notice that $\gamma'(t) = 0$, $t > t_\delta$. The evaluation routine SPVALUE will not yield this value so logic in the procedure *f_bc* assigns $\gamma'(t) = 0$, $t > t_\delta$.

```
FUNCTION f_ut,  npde,  x,  t,  u,  ux,  uxx
    ; Define the PDE
    ut  =  uxx
    RETURN, ut
END

PRO f_bc,  npde,  x,  t,  alpha, beta, gammap
COMMON ex1_pde, first, ppoly
    first  =  1
    alpha  =  FLTARR(npde)
    beta  =  FLTARR(npde)
    gammap  =  FLTARR(npde)
    delta  =  0.09
    ; Compute interpolant first time only
    IF (first EQ 1) THEN BEGIN
        first  =  0
        ppoly  =  CSINTERP([0.0,  delta],  [1.0,  0.1], $
            ileft = 1,  left = 0.0,  iright = 1,  right = 0.0)
    END
    ; Define the boundary conditions.
    IF (x EQ 0.0) THEN  BEGIN
        alpha(0)  =  1.0
        beta(0)  =  0.0
        gammap(0)  =  0.0
    ; If in the boundary layer, compute nonzero gamma prime
        IF (t LE delta) THEN gammap(0)  =  $
                  SPVALUE(t,  ppoly,  xderiv  =  1)
```

```
    END ELSE BEGIN
    ; These are for x = 1
        alpha(0)  =  0.0
        beta(0)  =  1.0
        gammap(0)  =  0.0
    END
    RETURN
END

COMMON ex1_pde, first, ppoly
npde  =  1
nx  =  8
nstep  =  10
    ; Set breakpoints and initial conditions
xbreak  =  FINDGEN(nx)/(nx - 1)
y  =  FLTARR(npde, nx)
y(*)  =  1.0
    ; Initialize the solver
t  =  FINDGEN(nstep)/(nstep) + 0.1
t  =  [0.0, t*t]
    ; Solve the problem
res  =  PDE_MOL(t, y,  xbreak,  'f_ut',  'f_bc')
num = INDGEN(8) + 1
    ; Print results at current t_i=t_i+1
FOR i = 1, 10 DO BEGIN
    PRINT,  'solution at t = ',  t(i)
    PRINT, num, Format = "(8I7)"
    PM,  res(0,  *,  i), Format = "(8F7.4)"
END
solution at t =      0.0100000
       1       2       3       4       5       6       7       8
 0.9691 0.9972 0.9999 1.0000 1.0000 1.0000 1.0000 1.0000
solution at t =      0.0400000
       1       2       3       4       5       6       7       8
 0.6247 0.8708 0.9624 0.9908 0.9981 0.9997 1.0000 1.0000
```

```
solution at t =      0.0900000
       1       2       3       4       5       6       7       8
 0.1000 0.4602 0.7169 0.8671 0.9436 0.9781 0.9917 0.9951
solution at t =       0.160000
       1       2       3       4       5       6       7       8
 0.1000 0.3130 0.5071 0.6681 0.7893 0.8708 0.9168 0.9315
solution at t =       0.250000
       1       2       3       4       5       6       7       8
 0.1000 0.2567 0.4045 0.5354 0.6428 0.7224 0.7710 0.7874
solution at t =       0.360000
       1       2       3       4       5       6       7       8
 0.1000 0.2176 0.3292 0.4292 0.5125 0.5751 0.6139 0.6270
solution at t =       0.490000
       1       2       3       4       5       6       7       8
 0.1000 0.1852 0.2661 0.3386 0.3992 0.4448 0.4731 0.4827
solution at t =       0.640000
       1       2       3       4       5       6       7       8
 0.1000 0.1588 0.2147 0.2648 0.3066 0.3381 0.3577 0.3643
solution at t =       0.810000
       1       2       3       4       5       6       7       8
 0.1000 0.1387 0.1754 0.2083 0.2358 0.2565 0.2694 0.2738
solution at t =        1.00000
       1       2       3       4       5       6       7       8
 0.1000 0.1242 0.1472 0.1678 0.1850 0.1980 0.2060 0.2087
```

Example 2

Here, Problem C is solved from Sincovec and Madsen (1975). The equation is of diffusion-convection type with discontinuous coefficients. This problem illustrates a simple method for programming the evaluation routine for the

derivative, u_t. Note that the weak discontinuities at x = 0.5 are not evaluated in the expression for u_t. The problem is defined as

$$u_t = \partial u / \partial t = \partial / \partial x\left(D(x)\partial u / \partial x\right) - v(x)\partial u / \partial x$$
$$x \in [0, 1], t > 0$$

$$D(x) = \begin{cases} 5 & \text{if } 0 \le x < 0.5 \\ 1 & \text{if } 0.5 < x \le 1.0 \end{cases}$$

$$v(x) = \begin{cases} 1000.0 & \text{if } 0 \le x < 0.5 \\ 1 & \text{if } 0.5 < x \le 1.0 \end{cases}$$

$$u(x, 0) = \begin{cases} 1 \text{ if } x = 0 \\ 0 \text{ if } x > 0 \end{cases}$$

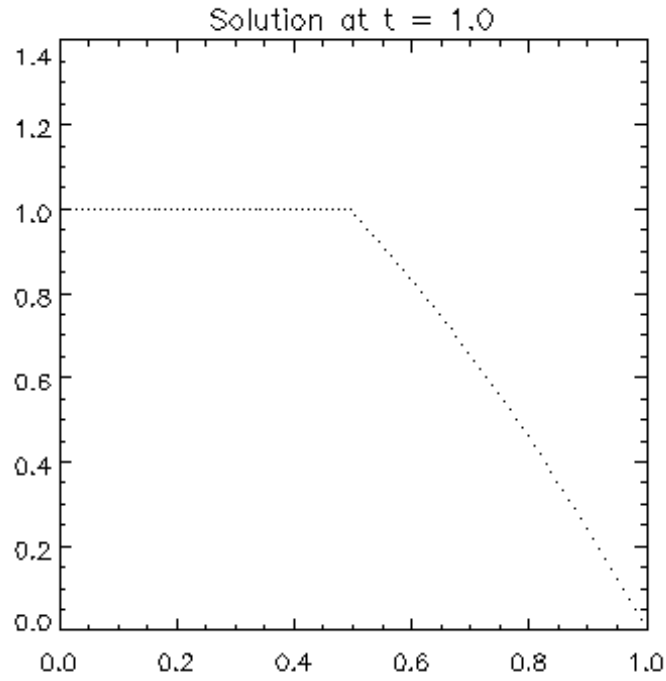
$$u(0, t) = 1, \quad u(1, t) = 0$$

```
FUNCTION f_ut,  npde,  x,  t,  u,  ux,  uxx
   ; Define the PDE
   ut  =  FLTARR(npde)
   IF (x LE 0.5) THEN BEGIN
   d  =  5.0
   v  =  1000.0
   END ELSE BEGIN
   d  =  1.0
   v  =  1.0
   END
   ut(0)  =  d*uxx(0) - v*ux(0)
   RETURN, ut
END

PRO f_bc,  npde,  x,  t,  alpha, beta, gammap
   ; Define the Boundary Conditions
```

```
    alpha   =   FLTARR(npde)
    beta   =   FLTARR(npde)
    gammap   =   FLTARR(npde)
    alpha(0)   =   1.0
    beta(0)   =   0.0
    gammap(0)   =   0.0
    RETURN
END

npde   =   1
nx   =   100
nstep   =   10
    ; Set breakpoints and initial conditions
xbreak   =   FINDGEN(nx)/(nx - 1)
y   =   FLTARR(npde, 100)
y(*)   =   0.0
y(0)   =   1.0
    ; Initialize the solver
mach   =   MACHINE(/FLOAT)
tol   =   SQRT(mach.MAX_REL_SPACE)
hinit   =   0.01*tol
PRINT, "tol = ", tol, " and hinit = ", hinit
t   =   [0.0, FINDGEN(nstep)/(nstep)   +   0.1]
    ; Solve the problem
res   =   PDE_MOL(t,   y,   xbreak,   'f_ut',   'f_bc', $
                      tolerance   =   tol,   hinit   =   hinit)
    ; Plot results at current t_i=t_i+1
PLOT, xbreak, res(0,*,10), psym = 3, yrange=[0 , 1.25], $
          title = "Solution at t = 1.0"
```



Example 3

In this example, using PDE_MOL, the linear normalized diffusion PDE $u_t = u_{xx}$ is solved but with an optional use that provides values of the derivatives, u_x, of the initial data. Due to errors in the numerical derivatives computed by spline interpolation, more precise derivative values are required when the initial data is $u(x, 0) = 1 + \cos[(2n - 1)\pi x]$, $n > 1$. The boundary conditions are "zero flux" conditions
$u_x(0, t) = u_x(1, t) = 0$ for $t > 0$. Note that the initial data is compatible with these end conditions since the derivative function

$$u_x(x,0) = \frac{du(x,0)}{dx} = -(2n-1)\pi \sin\left[(2n-1)\pi x\right]$$

vanishes at $x = 0$ and $x = 1$.

This optional usage signals that the derivative of the initial data is passed by the user.

```
FUNCTION f_ut,  npde,  x,  t,  u,  ux,  uxx
    ; Define the PDE
    ut  =  fltARR(npde)
    ut(0)  =  uxx(0)
    RETURN, ut
END

PRO f_bc,  npde,  x,  t,  alpha, beta, gammap
    ; Define the boundary conditions
    alpha  =  FLTARR(npde)
    beta  =  FLTARR(npde)
    gammap  =  FLTARR(npde)
    alpha(0)  =  0.0
    beta(0)  =  1.0
    gammap(0)  =  0.0
    RETURN
END

npde  =  1
nx  =  10
nstep  =  10
arg  =  9.0*!Pi
    ; Set breakpoints and initial conditions
xbreak  =  FINDGEN(nx)/(nx - 1)
y  =  FLTARR(npde,  nx)
y(0, *)  =  1.0 + COS(arg*xbreak)
di  =  y
di(0, *)  =  -arg*SIN(arg*xbreak)
    ; Initialize the solver
mach  =  MACHINE(/FLOAT)
tol = SQRT(mach.MAX_REL_SPACE)
t  =  [FINDGEN(nstep + 1)*(nstep*0.001)/(nstep)]
    ; Solve the problem
```

```
res  =  PDE_MOL(t,  y,  xbreak,  'f_ut',  'f_bc', $
                  Tolerance = tol,  Deriv_Init = di)
   ; Print results at every other t_i=t_i+1
FOR i = 2, 10, 2 DO BEGIN
   PRINT,  'solution at t = ',  t(i)
   PM,  res(0, *, i), Format = "(10F10.4)"
   PRINT,  'derivative at t = ',  t(i)
   PM,  di(0, *, i)
   PRINT
END
solution at t =     0.00200000
     1.2329     0.7671     1.2329     0.7671     1.2329
     0.7671     1.2329     0.7671     1.2329     0.7671
derivative at t =     0.00200000
       0.00000   9.58505e-07   7.96148e-09   1.25302e-06
 -1.61002e-07   1.91968e-06 -1.60244e-06   3.85856e-06
 -4.83314e-06   2.02301e-06

solution at t =     0.00400000
     1.0537     0.9463     1.0537     0.9463     1.0537
     0.9463     1.0537     0.9463     1.0537     0.9463
derivative at t =     0.00400000
       0.00000   6.64098e-07 -5.12883e-07   8.55131e-07
 -6.11177e-07 -2.76893e-06   7.84288e-08   2.97113e-06
 -2.32777e-07   2.02301e-06

solution at t =     0.00600000
     1.0121     0.9879     1.0121     0.9879     1.0121
     0.9879     1.0121     0.9879     1.0121     0.9879
derivative at t =     0.00600000
       0.00000   7.42109e-07 -5.29244e-08 -1.98559e-07
 -1.19702e-06 -8.66795e-07   1.17180e-07   7.09625e-07
  4.31432e-07   2.02301e-06
```

```
solution at t =     0.00800000
     1.0027     0.9973     1.0027     0.9973     1.0027
     0.9973     1.0027     0.9973     1.0027     0.9973
derivative at t =     0.00800000
       0.00000   3.56892e-07 -3.80790e-07 -9.99308e-07
 -1.96765e-07   7.72356e-07   8.50576e-08   1.11979e-07
  4.74838e-07   2.02301e-06

solution at t =      0.0100000
     1.0008     0.9992     1.0008     0.9992     1.0008
     0.9992     1.0008     0.9992     1.0008     0.9992
derivative at t =      0.0100000
       0.00000   2.40533e-07 -4.27171e-07 -1.25933e-06
  3.60702e-08   6.42627e-07 -1.00818e-07   2.08207e-07
  1.12973e-06   2.02301e-06
```

Example 4

In this example, consider the linear normalized hyperbolic PDE, $u_{tt} = u_{xx}$, the "vibrating string" equation. This naturally leads to a system of first order PDEs. Define a new dependent variable $u_t = v$. Then, $v_t = u_{xx}$ is the second equation in the system. Take as initial data $u(x, 0) = \sin(\pi x)$ and $u_t(x, 0) = v(x, 0) = 0$. The ends of the string are fixed so $u(0, t) = u(1, t) = v(0, t) = v(1, t) = 0$. The exact solution to this problem is $u(x, t) = \sin(\pi x)\cos(\pi t)$. Residuals are computed at the output values of t for $0 < t \leq 2$. Output is obtained at 200 steps in increments of 0.01.

Even though the sample code PDE_MOL gives satisfactory results for this PDE, users should be aware that for *nonlinear problems*, "shocks" can develop in the solution. The appearance of shocks may cause the code to fail in unpredictable ways. See Courant and Hilbert (1962), pp 488-490, for an introductory discussion of shocks in hyperbolic systems.

```
FUNCTION f_ut,  npde,  x,  t,  u,  ux,  uxx
   ; Define the PDE
   ut  =  FLTARR(npde)
   ut(0)  =  u(1)
   ut(1)  =  uxx(0)
   RETURN, ut
```

```
END

PRO f_bc,  npde,  x,  t,  alpha, beta, gammap
   ; Define the boundary conditions
   alpha  =  FLTARR(npde)
   beta  =  FLTARR(npde)
   gammap  =  FLTARR(npde)
   alpha(0)  =  1
   alpha(1)  =  1
   beta(0)  =  0
   beta(1)  =  0
   gammap(0)  =  0
   gammap(1)  =  0
   RETURN
END

npde  =  2
nx  =  10
nstep  =  200
   ; Set breakpoints and initial conditions
xbreak  =  FINDGEN(nx)/(nx - 1)
y  =  FLTARR(npde,  nx)
y(0, *)  =  SIN(!Pi*xbreak)
y(1, *)  =  0
di  =  y
di(0, *)  =  !Pi*COS(!Pi*xbreak)
di(1, *)  =  0.0
   ; Initialize the solver
mach  =  MACHINE(/FLOAT)
tol  =  SQRT(mach.MAX_REL_SPACE)
t  =  [0.0, 0.01 + FINDGEN(nstep)*2.0/(nstep)]
   ; Solve the problem
u  =  PDE_MOL(t,  y,  xbreak,  'f_ut',  'f_bc', $
                  Tolerance  =  tol,  Deriv_Init  =  di)
```

```
err  =  0.0
pde_error  =  FLTARR(nstep)
FOR j  =  1,  N_ELEMENTS(t) - 1 DO BEGIN
   FOR i = 0, nx - 1 DO BEGIN
      err = (err) > (u(0, i, j) - $
               SIN(!Pi*xbreak(i))*COS(!Pi*t(j)))
   END
END
PRINT, "Maximum error in u(x, t) = ", err
Maximum error in u(x, t) =    0.000626385
```

POISSON2D Function

Solves Poisson's or Helmholtz's equation on a two-dimensional rectangle using a fast Poisson solver based on the HODIE finite-difference scheme on a uniform mesh.

Usage

result = POISSON2D(*rhs_pde*, *rhs_bc*, *coef_u*, *nx*, *ny*, *ax*, *bx*, *ay*, *by*, *bc_type*)

Input Parameters

rhs_pde — Scalar string specifying the name of the user-supplied function to evaluate the right-hand side of the partial differential equation at a scalar value *x* and scalar value *y*.

rhs_bc — Scalar string specifying the name of the user-supplied function to evaluate the right-hand side of the boundary conditions, on side *side*, at scalar value *x* and scalar value *y*. The value of *side* will be one of the following integer values:

Integer	Side
0	right side
1	bottom side
2	left side
3	top side

coef_u — Value of the coefficient of *u* in the differential equation.

nx — Number of grid lines in the *x*-direction. *nx* must be at least 4. See the *Discussion* section for further restrictions on *nx*.

ny — Number of grid lines in the *y*-direction. *ny* must be at least 4. See the *Discussion* section for further restrictions on *ny*.

ax — The value of *x* along the left side of the domain.

bx — The value of *x* along the right side of the domain.

ay — The value of *y* along the bottom of the domain.

by — The value of *y* along the top of the domain.

bc_type — One-dimensional array of size 4 indicating the type of boundary condition on each side of the domain or that the solution is periodic. The sides are numbered as follows:

Array	Side	Location
bc_type(0)	right	$x = bx$
bc_type(1)	bottom	$y = ay$
bc_type(2)	left	$x = ax$
bc_type(3)	top	$y = by$

The three possible boundary condition types are as follows:

Type	Condition
bc_type(*i*) = 1	Dirichlet condition. Value of *u* is given.
bc_type(*i*) = 2	Neuman condition. Value of *du*/*dx* is given (on the right or left sides) or *du*/*dy* (on the bottom or top of the domain).
bc_type(*i*) = 3	Periodic condition.

Returned Value

result — Two-dimensional array of size *nx* by *ny* containing solution at the grid points.

Input Keywords

Double — If present and nonzero, double precision is used.

Order — Order of accuracy of the finite-difference approximation. It can be either 2 or 4.

Default: *Order* = 4

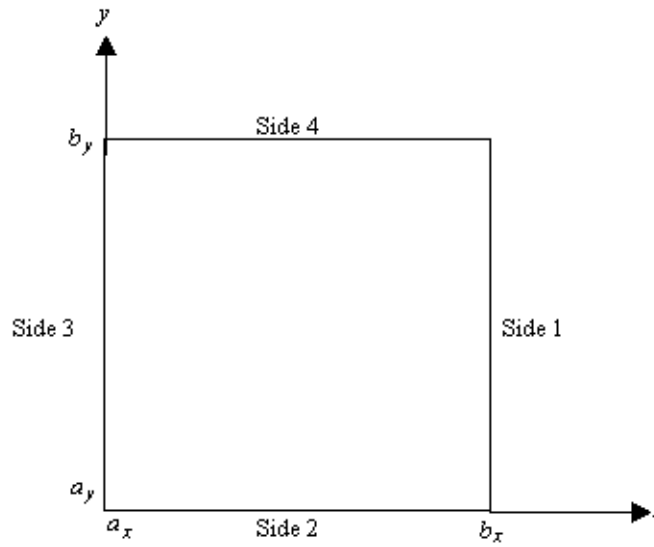
Discussion

Let $c = coef\_u$, $a_x = ax$, $b_x = bx$, $a_y = ay$, $b_y = by$, $n_x = nx$ and $n_y = ny$.

POISSON2D is based on the code HFFT2D by Boisvert (1984). It solves the equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + cu = p$$

on the rectangular domain $(a_x, b_x) \times (a_y, b_y)$ with a user-specified combination of Dirichlet (solution prescribed), Neumann (first-derivative prescribed), or periodic boundary conditions. The sides are numbered clockwise, starting with the right side.



When $c = 0$ and only Neumann or periodic boundary conditions are prescribed, then any constant may be added to the solution to obtain another solution to the problem. In this case, the solution of minimum ∞-norm is returned.

The solution is computed using either a second-or fourth-order accurate finite-difference approximation of the continuous equation. The resulting system of linear algebraic equations is solved using fast Fourier transform techniques. The

algorithm relies on the fact that $n_x - 1$ is highly composite (the product of small primes). For details of the algorithm, see Boisvert (1984). If $n_x - 1$ is highly composite then the execution time of POISSON2D is proportional to $n_x n_y \log_2 n_x$. If evaluations of $p(x, y)$ are inexpensive, then the difference in running time between *Order* = 2 and *Order* = 4 is small.

The grid spacing is the distance between the (uniformly spaced) grid lines. It is given by the formulas $hx = (bx - ax)/(nx - 1)$ and $hy = (by - ay)/(ny - 1)$. The grid spacings in the *x* and *y* directions must be the same, i.e., *nx* and *ny* must be such that *hx* is equal to *hy*. Also, as noted above, *nx* and *ny* must be at least 4. To increase the speed of the fast Fourier transform, $nx - 1$ should be the product of small primes. Good choices are 17, 33, and 65.

If *-coef_u* is nearly equal to an eigenvalue of the Laplacian with homogeneous boundary conditions, then the computed solution might have large errors.

Example

In this example, the equation

$$\frac{\partial^2 u}{\partial x_2} + \frac{\partial^2 u}{\partial y^2} + 3u = -2\sin(x + 2y) + 16e^{2x+3y}$$

with the boundary conditions

$$\frac{\partial u}{\partial y} = 2\cos(x + 2y) + 3e^{2x+3y}$$

on the bottom side and

$$m = \sin(x + 2y) + e^{2x+3y})$$

on the other three sides is solved. The domain is the rectangle $[0, 1/4] \times [0, 1/2]$. The output of POISSON2D is a 17×33 table of values. The functions SPVALUE are used to print a different table of values.

```
FUNCTION rhs_pde,  x,  y
   ; Define the right side of the PDE
   f  =  (-2.0*SIN(x + 2.0*y) + 16.0*EXP(2.0*x + 3.0*y))
   RETURN,  f
END

FUNCTION rhs_bc,  side,  x,  y
   ; Define the boundary conditions
   IF (side EQ 1) THEN $
   ; Bottom side
   f  =  2.0*COS(x + 2.0*y) + 3.0*EXP(2.0*x + 3.0*y) $
   ELSE $
   ; All other sides, 0, 2, 3
   f  =  SIN(x + 2.0*y) + EXP(2.0*x + 3.0*y)
   RETURN,  f
END

PRO print_results, x, y, utable
   FOR j  =  0, 4 DO FOR i  =  0, 4 DO  $
      PRINT,  x(i),  y(j),  utable(i, j),  $
      ABS(utable(i,  j) - SIN(x(i) + 2.0*y(j)) - $
               EXP(2.0*x(i) + 3.0*y(j)))
END

nx  =  17
nxtable  =  5
ny  =  33
nytable  =  5
   ; Set rectangle size
ax  =  0.0
bx  =  0.25
ay  =  0.0
by  =  0.5
   ; Set boundary conditions
bc_type  =  [1, 2, 1, 1]
```

```
    ; Coefficient of u
coef_u  =  3.0
    ; Solve the PDE
u  =  POISSON2D('rhs_pde', 'rhs_bc', coef_u, nx, ny, ax, $
                  bx, ay, by, bc_type)
    ; Set up for interpolation
xdata  =  ax + (bx - ax)*FINDGEN(nx)/(nx - 1)
ydata  =  ay + (by - ay)*FINDGEN(ny)/(ny - 1)
    ; Compute interpolant
sp  =  BSINTERP(xdata,  ydata,  u)
x  =  ax + (bx - ax)*FINDGEN(nxtable)/(nxtable - 1)
y  =  ay + (by - ay)*FINDGEN(nytable)/(nytable - 1)
utable  =  SPVALUE(x, y, sp)
    ; Print computed answer and absolute on nxtabl by nytabl grid
PRINT,"          X              Y              U          Error"
print_results, x, y, utable
          X              Y              U          Error
       0.00000        0.00000        1.00000        0.00000
     0.0625000        0.00000        1.19560   4.88758e-06
      0.125000        0.00000        1.40869   7.39098e-06
      0.187500        0.00000        1.64139   4.88758e-06
      0.250000        0.00000        1.89613   1.19209e-07
       0.00000       0.125000        1.70240   1.19209e-07
     0.0625000       0.125000        1.95615   6.55651e-06
      0.125000       0.125000        2.23451   9.53674e-06
      0.187500       0.125000        2.54067   6.67572e-06
      0.250000       0.125000        2.87830        0.00000
       0.00000       0.250000        2.59643   4.76837e-07
     0.0625000       0.250000        2.93217   9.05991e-06
      0.125000       0.250000        3.30337   1.31130e-05
      0.187500       0.250000        3.71482   8.82149e-06
      0.250000       0.250000        4.17198   2.38419e-07
       0.00000       0.375000        3.76186   2.38419e-07
     0.0625000       0.375000        4.21634   9.05991e-06
```

```
  0.125000      0.375000       4.72261   1.31130e-05
  0.187500      0.375000       5.28776   8.58307e-06
  0.250000      0.375000       5.91989   4.76837e-07
   0.00000      0.500000       5.32316   4.76837e-07
 0.0625000      0.500000       5.95199       0.00000
  0.125000      0.500000       6.65687   4.76837e-07
  0.187500      0.500000       7.44826       0.00000
  0.250000      0.500000       8.33804   1.43051e-06
```

6

CHAPTER

Transforms

Contents of Chapter

Introduction

Fast Fourier Transforms

A fast Fourier transform (FFT) is simply a discrete Fourier transform that is computed efficiently. Basically, the straightforward method for computing the Fourier transform takes approximately n^2 operations, where n is the number of points in the transform, while the FFT (which computes the same values) takes approximately $n\log n$ operations. The algorithms in this chapter are modeled after the Cooley-Tukey (1965) algorithm. Hence, these functions are most efficient for integers that are highly composite, that is, integers that are a product of the small primes 2, 3, and 5.

For the function FFTCOMP, there is a corresponding initialization function. Use this function only when repeatedly transforming one-dimensional sequences of the same data type and length. In this situation, the initialization function computes the initial setup once; subsequently, the user calls the main function with the appropriate keyword. This may result in substantial computational savings. For more information on the use of these functions, consult the documentation under the appropriate function name. In addition to the one-dimensional transformation described above, the function FFTCOMP also can be used to compute a complex two-dimensional FFT and its inverse.

Continuous Versus Discrete Fourier Transform

There is a close connection between the discrete Fourier transform and the continuous Fourier transform. The continuous Fourier transform is defined by Brigham (1974) as follows:

$$\hat{f}(\omega) = (Ff)(\omega) = \int_{-\infty}^{\infty} f(t)e^{-2\pi i\omega t}dt$$

Begin by making the following approximation:

$$\hat{f}(\omega) \approx \int_{-T/2}^{T/2} f(t)e^{-2\pi i\omega t}dt$$

$$= \int_{0}^{T} f(t - T/2)e^{-2\pi i\omega(t - T/2)}dt$$

$$= e^{\pi i\omega T}\int_{0}^{T} f(t - T/2)e^{-2\pi i\omega t}dt$$

If the last integral approximated using the rectangle rule with spacing $h = T / n$, the result is given below:

$$\hat{f}(\omega) \approx e^{\pi i\omega T}h\sum_{k=0}^{n-1} e^{-2\pi i\omega kh} f(kh - T/2)$$

Finally, setting $\omega = j / T$ for $j = 0, ..., n - 1$ yields

$$\hat{f}(j/T) \approx e^{\pi ij}h\sum_{k=0}^{n-1} e^{-2\pi ij(k/n)}f(kh - T/2) = -1^{j}h\sum_{k=0}^{n-1} e^{-2\pi ij(k/n)}f_{k}^{h}$$

where the vector $f^h = (f(-T/2), ..., f((n-1)h - T/2))$. Thus, after scaling the components by $(-1)^j h$, the discrete Fourier transform as computed in FFTCOMP (with input f^h) is related to an approximation of the continuous Fourier transform by the above formula.

If the function f is expressed as a function, then the continuous Fourier transform

$\hat{f}$

can be approximated using the PV-WAVE:IMSL Mathematics function INTFCN to compute a Fourier transform as described on page 193 in Chapter 4.

FFTCOMP Function

Computes the discrete Fourier transform of a real or complex sequence. Using keywords, a real-to-complex transform or a two-dimensional complex Fourier transform can be computed.

Usage

result = FFTCOMP(*a*)

Input Parameters

a — Array containing the periodic sequence.

Returned Value

result — The transformed sequence. If *A* is one-dimensional, the type of *A* determines whether the real or complex transform is computed, where *A* is array *a*. If *A* is two-dimensional, the complex transform is always computed.

Input Keywords

Cosine — If present and nonzero, then FFTCOMP computes the discrete Fourier cosine transformation of an even sequence

Sine — If present and nonzero, then FFTCOMP computes the discrete Fourier sine transformation of an odd sequence

Double — If present and nonzero, double precision is used.

Complex — If present and nonzero, the complex transform is computed. If *A* is complex, this keyword is not required to ensure that a complex transform is computed. If *A* is real, it is promoted to complex internally.

Backward — If present and nonzero, the backward transform is computed. See the *Discussion* section below for more details on this option.

Init_Params — Array containing parameters used when computing a one-dimensional FFT. If FFTCOMP is used repeatedly with arrays of the same length and data type, it is more efficient to compute these parameters only once with a call to function FFTINIT.

Discussion

The default action of the function FFTCOMP is to compute the FFT of an array *A*, with the type of FFT performed dependent upon the data type of the input array *A*. (If *A* is a one-dimensional real array, the real FFT is computed; if *A* is a one-dimensional complex array, the complex FFT is computed; and if *A* is a two-dimensional real or complex array, the complex FFT is computed.) If the complex FFT of a one-dimensional real array is desired, keyword *Complex* should be specified. The keywords *Sine* and *Cosine* allow FFTCOMP to be used to compute the discrete Fourier sine or cosine transformation of a one dimensional real array. The remainder of this section is divided into separate discussions of the various uses of FFTCOMP.

Case 1: One-dimensional Real FFT

If *A* is one-dimensional and real, the function FFTCOMP computes the discrete Fourier transform of a real array of length n = N_ELEMENTS (*a*). The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when n is a product of small prime factors. If n satisfies this condition, then the computational effort is proportional to $n\log n$.

By default, FFTCOMP computes the forward transform. If n is even, the forward transform is as follows:

$$q_{2m-1} = \sum_{k=0}^{n-1} p_k \cos\frac{2\pi km}{n}$$

$$q_{2m} = -\sum_{k=0}^{n-1} p_k \sin\frac{2\pi km}{n}$$

$$q_0 = \sum_{k=0}^{n-1} p_k$$

If n is odd, q_m is defined as above for m from 1 to $(n - 1) / 2$.

Let f be a real-valued function of time. Suppose f is sampled at n equally spaced time intervals of length Δ seconds starting at time t_0:

$$p_i = f(t_0 + i\Delta) \qquad i = 0, 1, ..., n - 1$$

Assume that n is odd for the remainder of the discussion for the case in which A is real. Function FFTCOMP treats this sequence as if it were periodic of period n. In particular, it assumes that $f(t_0) = f(t_0 + n\Delta)$. Hence, the period of the function is assumed to be $T = n\Delta$. The above transform is inverted for the following:

$$p_m = \frac{1}{n}\left[q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1}\cos\frac{2\pi km}{n} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2}\sin\frac{2\pi km}{n}\right]$$

This formula can be interpreted in the following manner: The coefficients q produced by FFTCOMP determine an interpolating trigonometric polynomial to the data. That is, if the equations are defined as

$$g(t) = \frac{1}{n}\left[q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1}\cos\frac{2\pi k(t-t_0)}{n\Delta} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2}\sin\frac{2\pi k(t-t_0)}{n\Delta}\right]$$

$$g(t) = \frac{1}{n}\left[q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1}\cos\frac{2\pi k(t-t_0)}{T} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2}\sin\frac{2\pi k(t-t_0)}{T}\right]$$

then the result is as follows:

$$f(t_0 + (i - 1)\,\Delta) = g(t_0 + (i - 1)\,\Delta)$$

Now suppose the dominant frequencies are to be obtained. Form the array *P* of length $(n + 1) / 2$ as follows:

$$P_0 = |q_0|$$

$$P_k = \sqrt{q_{2k}^2 + q_{2k-1}^2} \qquad k = 1, 2, \ldots, (n-1)/2$$

These numbers correspond to the energy in the spectrum of the signal. In particular, P_k corresponds to the energy level at the following frequency:

$$\frac{k}{T} = \frac{k}{n\Delta} \qquad k = 0, 1, \ldots, \frac{n-1}{2}$$

Furthermore, note that there are only

$$(n + 1)/2 \approx T/(2\Delta)$$

resolvable frequencies when *n* observations are taken. This is related to the Nyquist phenomenon, which is induced by discrete sampling of a continuous signal. Similar relations hold for the case when *n* is even.

If keyword *Backward* is specified, the backward transform is computed. If *n* is even, the backward transform is as follows:

$$q_m = p_0 + (-1)^{m+1} p_{n-1} + 2 \sum^{n/2-1} p_{2k+1} \cos\frac{2\pi km}{n} - 2 \sum^{n/2-2} p_{2k+2} \sin\frac{2\pi km}{n}$$

If *n* is odd, the following is true:

$$q_m = p_0 + 2 \sum_{k=0}^{(n-3)/2} p_{2k+1} \cos\frac{2\pi km}{n} - 2 \sum_{k=0}^{(n-3)/2} p_{2k+2} \sin\frac{2\pi km}{n}$$

The backward Fourier transform is the unnormalized inverse of the forward Fourier transform.

The FFTCOMP function is based on the real FFT in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

Case 2: One-dimensional Complex FFT

If *A* is one-dimensional and complex, function FFTCOMP computes the discrete Fourier transform of a complex array of size *n* = N_ELEMENTS (*a*). The method used is a variant of the Cooley Tukey algorithm, which is most efficient when *n* is a product of small prime factors. If *n* satisfies this condition, the computational effort is proportional to *n*log*n*.

By default, FFTCOMP computes the forward transform as in the equation below.

$$q_j = \sum_{n-1} p_m e^{(-2\pi imj)/n}$$

Note, the Fourier transform can be inverted as follows:

$$p_m = \frac{1}{n} \sum_{n-1} q_j e^{2\pi ij(m/n)}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, you have coefficients for a trigonometric interpolating polynomial to the data.

If keyword *Backward* is used, the following computation is performed:

$$q_j = \sum_{n-1} p_m e^{2\pi im(j/n)}$$

Furthermore, the relation between the forward and backward transforms is that they are unnormalized inverses of each other. In other words, the following code fragment begins with an array *p* and concludes with an array $p_2 = np$:

```
q  = FFTCOMP(p)
p2 = FFTCOMP(q, /Backward)
```

Case 3: Two-dimensional FFT

If *A* is two-dimensional and real or complex, function FFTCOMP computes the discrete Fourier transform of a two-dimensional complex array of size *n* x *m* where *n* = N_ELEMENTS (*a* (*, 0)) and *m* = N_ELEMENTS (*a* (0, *)). The method used is a variant of the Cooley-Tukey algorithm, which is most efficient

when both *n* and *m* are a product of small prime factors. If *n* and *m* satisfy this condition, then the computational effort is proportional to *nm*log*nm*.

By default, given a two-dimensional array, FFTCOMP computes the forward transform as in the following equation:

$$q_{jk} = \sum_{n-1}^{m-1} \sum p_{st} e^{-2\pi ijs/n} e^{-2\pi ikt/m}$$

Note, the Fourier transform can be inverted as follows:

$$p_{jk} = \frac{1}{nm} \sum_{n-1}^{m-1} \sum q_{st} e^{2\pi ijs/n} e^{2\pi ikt/m}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, you have the coefficients for a trigonometric interpolating polynomial to the data.

If keyword *Backward* is used, the following computation is performed:

$$p_{jk} = \sum_{n-1}^{m-1} \sum q_{st} e^{2\pi ijs/n} e^{2\pi ikt/m}$$

Case 4: Cosine Transform of a Real Sequence

If the keyword *Cosine* is present and nonzero, the function FFTCOMP computes the discrete Fourier cosine transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when $N - 1$ is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to $N \log N$. Specifically, given an *N*-vector p, FFTCOMP returns in q

$$q_m = 2 \sum_{n=1}^{N-2} p_n \sin\left(\frac{mn\pi}{N-1}\right) + s_0 + s_{N-1}(-1)^m$$

where p = array *a* and q = result.

Finally, note that the Fourier cosine transform is its own (unnormalized) inverse.

Case 5: Sine Transform of a Real Sequence

If the keyword *Sine* is present and nonzero, the function FFTCOMP computes the discrete Fourier sine transform of a real vector of size *N*. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when $N + 1$ is a product of small prime factors. If *N* satisfies this condition, then the computational effort is proportional to $N \log N$. Specifically, given an *N*-vector p, FFTCOMP returns in q

$$q_m = 2\sum_{n=0}^{N-1} p_n \sin\left(\frac{(m+1)(n+1)\pi}{N+1}\right)$$

where p = array *a* and q = result.

Finally, note that the Fourier sine transform is its own (unnormalized) inverse.

Example 1

In this example, a pure cosine wave is used as a data array, and its Fourier series is recovered. The Fourier series is an array with all components zero except at the appropriate frequency where it has an *n*/2.

```
n = 7
    ; Fill up the data array with a pure cosine wave.
p = COS(FINDGEN(n) * 2 * !Pi/n)
PM, p
      1.00000
      0.623490
     -0.222521
     -0.900969
     -0.900969
     -0.222521
      0.623490
q = FFTCOMP(p)
    ; Call FFTCOMP to compute the FFT.
PM, q, Format = '(f8.3)'
    ; Output results.
    0.000
    3.500
    0.000
   -0.000
```

```
  -0.000
   0.000
  -0.000
```

Example 2: Resolving Dominant Frequencies

The following procedure demonstrates how the FFT can be used to resolve the dominant frequency of a signal. Call FFTCOMP with a data vector of length *n* = 15, filled with pure, exponential signals of increasing frequency and decreasing strength. Using the computed FFT, the relative strength of the frequencies is resolved. It is important to note that for an array of length *n*, at most $(n + 1)/2$ frequencies can be resolved using the computed FFT.

```
PRO power_spectrum

n = 15
    ; Define the length of the signal.

num_freq = n/2 + (n MOD 2)
z = COMPLEX(0, FINDGEN(n) * 2 * !Pi/n)
p = COMPLEXARR(n)

FOR i = 0, num_freq - 1 DO $
    p = p + EXP(i * z)/(i + 1)
        ; Fill up the data array.

q = FFTCOMP(p)
    ; Compute the FFT.

power = FLTARR(num_freq)

IF ((n MOD 2) EQ 0) THEN BEGIN
    power(0) = ABS(q(0))^2
    FOR i = 1,(num_freq - 2) DO $
           power(i) = q(i) * CONJ(q(i)) + $
           q(n-i-1) * CONJ(q(n-i-1))
           power(num_freq - 1) = q(num_freq - 1) * $
           CONJ(q(num_freq - 1))
    END
        ; Determine the strengths of the frequencies. The method is
        ; dependent upon whether n is even or odd.

IF ((n MOD 2) EQ 1) THEN BEGIN
    FOR i = 1,(num_freq - 1) DO power(i) = $
           q(i)^2 + q(n - i)^2
    power(0) = q(0)^2
END

PRINT, '    frequency   strength' &$
```

```
PRINT, '    ---------   --------' &$
FOR i = 0,7 DO PRINT, i, power(i)
    ; Display frequencies and strengths.
END
frequency   strength
---------   --------
     0        225.000
     1        56.2500
     2        25.0000
     3        14.0625
     4        9.00000
     5        6.25000
     6        4.59183
     7        3.51562
```

Example 3: Computing a Two-dimensional FFT

In this example, the forward transform of a two-dimensional matrix followed by the backward transform is computed. Notice that the process of computing the forward transform followed by the backward transform multiplies the entries of the original matrix by the product of the lengths of the two dimensions.

```
n = 4
m = 5
p = COMPLEXARR(n, m)
FOR i = 0, n - 1 DO BEGIN &$
    z = COMPLEX(0, 2 * i * 2 * !Pi/n) &$
    FOR j = 0, m - 1 DO BEGIN &$
           w = COMPLEX(0, 5 * j * 2 * !Pi/m) &$
           p(i, j) = EXP(z) * EXP(w) &$
    ENDFOR &$
ENDFOR
q = FFTCOMP(p)
p2 = FFTCOMP(q, /Backward)
format = "(4('(',f6.2,',',f5.2,')',2x))"
PM, p, Format = format, Title = 'p'
p
( 1.0,  0.0)( 1.0,  0.0)( 1.0,  0.0)( 1.0,  0.0)
( 1.0,  0.0)(-1.0,-0.0)(-1.0,-0.0)(-1.0,-0.0)
(-1.0,-0.0)(-1.0,-0.0)( 1.0,  0.0)( 1.0,  0.0)
( 1.0,  0.0)( 1.0,  0.0)( 1.0,  0.0)(-1.0,-0.0)
```

```
(-1.0,-0.0)(-1.0,-0.0)(-1.0,-0.0)(-1.0,-0.0)
PM, q, Format = format, Title = 'q = FFTCOMP(p)'
q = FFTCOMP(p)
( 0.0, 0.0)(-0.0, 0.0)( 0.0, 0.0)(-0.0, 0.0)
( 0.0, 0.0)(-0.0,-0.0)( 0.0,-0.0)( 0.0,-0.0)
( 0.0, 0.0)(-0.0, 0.0)(20.0, 0.0)(-0.0,-0.0)
(-0.0,-0.0)( 0.0,-0.0)( 0.0,-0.0)( 0.0,-0.0)
( 0.0, 0.0)(-0.0, 0.0)(-0.0,-0.0)(-0.0,-0.0)
PM, p2, Format = format, Title = 'p2 = FFTCOMP(q, /Backward)'
p2 = FFTCOMP(q, /Backward)
( 20., 0.)( 20., 0.)( 20., 0.)( 20., 0.)
( 20., 0.)(-20.,-0.)(-20.,-0.)(-20.,-0.)
(-20.,-0.)(-20.,-0.)( 20., 0.)( 20., 0.)
( 20., 0.)( 20., 0.)( 20., 0.)(-20.,-0.)
(-20.,-0.)(-20.,-0.)(-20.,-0.)(-20.,-0.)
```

FFTINIT Function

Computes the parameters for a one-dimensional FFT to be used in function FFTCOMP with keyword *Init_Params*.

Usage

result = FFTINIT(*n*)

Input Parameters

n — Length of the sequence to be transformed.

Returned Value

result — A one-dimensional array of length $2n + 15$ that can then be used by FFTCOMP when the optional parameter *Init_Params* is specified.

Input Keywords

Double — If present and nonzero, double precision is used and the returned array is double precision. This keyword does not have an effect if the initialization is being computed for a complex FFT.

Complex — If present and nonzero, the parameters for a complex transform are computed.

Sine — If present and nonzero, then parameters for a discrete Fourier cosine transformation are returned. See keyword *Sine* in function FFTCOMP.

Cosine — If present and nonzero, then parameters for a discrete Fourier cosine transformation are returned. See keyword *Sine* in function FFTCOMP.

Discussion

Function FFTINIT should be used when many calls are to be made to function FFTCOMP without changing the data type of the array and the length of the sequence. The default action of FFTINIT is to compute the parameters necessary for a real FFT. If parameters for a complex FFT are needed, keyword *Complex* should be specified.

The FFTINIT function is based on the routines RFFTI and RFFTI in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

Example

In this example, two distinct, real FFTs are computed by calling FFTINIT once, then calling function FFTCOMP twice.

```
n = 7
    ; Define the length of the signals.
init_params = FFTINIT(7)
    ; Initialize the parameters by calling FFTINIT.
FOR j = 0, 2 DO BEGIN $
   p = COS(j * FINDGEN(n) * 2 * !Pi/n) &$
   q = FFTCOMP(p, Init_Params = init_params)&$
   PM, 'p', 'q', &$
   Format = '(7x, a1, 10x, a1)' &$
   FOR i = 0, n - 1 DO PM, p(i), q(i), &$
          Format = '(f10.5, f10.2)' &$
ENDFOR
    ; For each pass through the loop, compute a real FFT of an array of
    ; length n and output both the original signal and the computed FFT.
       p          q
   1.00000       7.00
   1.00000       0.00
```

```
 1.00000       0.00
 1.00000       0.00
 1.00000       0.00
 1.00000      -0.00
 1.00000       0.00
    p          q
 1.00000       0.00
 0.62349       3.50
-0.22252       0.00
-0.90097      -0.00
-0.90097      -0.00
-0.22252       0.00
 0.62349      -0.00
    p          q
 1.00000      -0.00
-0.22252       0.00
-0.90097      -0.00
 0.62349       3.50
 0.62349      -0.00
-0.90097       0.00
-0.22252       0.00
```

CONVOL1D Function

Computes the discrete convolution of two one dimensional arrays.

Usage

result = CONVOL1D(*x*, *y*)

Input Parameters

x — One-dimensional array.

y — One-dimensional array.

Returned Value

result — A one-dimensional array containing the discrete convolution of *x* and *y*.

Input Keywords

Direct — If present and nonzero, causes the computations to be done by the direct method instead of the FFT method regardless of the size of the vectors passed in.

Periodic — If present and nonzero, then a *circular convolution* is computed.

Discussion

The function CONVOL1D computes the discrete convolution of two sequences *x* and *y*.

Let n_x be the length of x, and n_y denote the length of y. If keyword *Periodic* is set, then $n_z = max\{n_x, n_y\}$, otherwise n_z is set to the smallest whole number, $n_z \geq n_x + n_y - 1$, of the form

$$n_z = 2^{\alpha}3^{\beta}5^{\gamma} \quad : \alpha, \beta, \gamma \text{ nonnegative integers.}$$

The arrays x and y are then zero-padded to a length n_z. Then, we compute

$$z_i = \sum_{j=0}^{n_z - 1} x_{i-j} y_j$$

where the index on x is interpreted as a nonnegative number between 0 and $n_z - 1$.

The technique used to compute the z_i's is based on the fact that the (complex discrete) Fourier transform maps convolution into multiplication. Thus, the Fourier transform of z is given by

$$\hat{z}(n) = \hat{x}(n)\hat{y}(n)$$

where the following equation is true.

$$\hat{z}(n) = \sum_{m=0}^{n_z - 1} z_m e^{-2\pi i m n / n_z}$$

The technique used here to compute the convolution is to take the discrete Fourier transform of x and y, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that n_z is the product of small primes if *Periodic* is set. If n_z is a product of small primes, then the computational effort will be proportional to $n_z \log(n_z)$. If *Periodic* is not set, then n_z is chosen to be a product of small primes.

We point out that if x and y are not complex, then no complex transforms of x or y are taken, since a real transforms can simulate the complex transform above. Such a strategy is six times faster and requires less space than when using the complex transform.

Example

In this example, CONVOL1D is used to compute simple moving-average digital filter plots of 5-point and 25-point moving average filters of noisy data are produced. Results are shown in figures *Figure 6-1* and *Figure 6-2*.

```
PRO Convol1d_ex1

RANDOMOPT, SET = 1234579L
        ; Set the random number seed.

ny = 100
```

```
t = FINDGEN(ny)/(ny-1)
y = SIN(2*!PI*t) + $
    .5*RANDOM(ny, /Uniform) -.25
        ; Define a 1-period sine wave with added noise.
FOR nfltr = 5, 25, 20 DO BEGIN
nfltr_str = strcompress(nfltr,/Remove_All)
    fltr = fltarr(nfltr)
    fltr(*) = 1./nfltr
        ; Define the NFLTR-point moving average array.
    z = CONVOL1D(fltr, y, /Periodic)
        ; Convolve the filter and the signal, using the keyword Periodic.
    PLOT, y, LINESTYLE = 1, TITLE = $
              nfltr_str+'-point Moving Average'
    OPLOT, shift(z, -nfltr/2)
ENDFOR
END
```

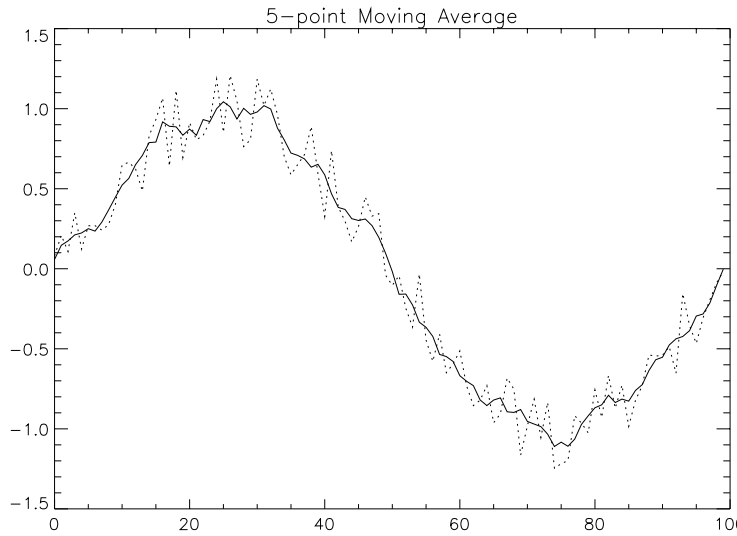


Figure 6-1 5-point moving average.

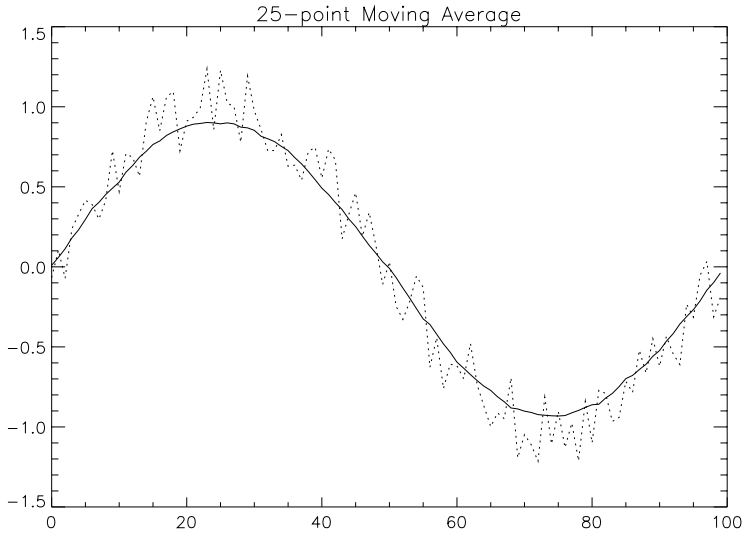


Figure 6-2 25-point moving average.

CORR1D Function

Compute the discrete correlation of two one-dimensional arrays.

Usage

result = CORR1D(*x*[, *y*])

Input Parameters

x— One-dimensional array.

y— (Optional) One-dimensional array.

Returned Value

result — A one-dimensional array containing the discrete convolution of *x* and *x*, or *x* and *y* if *y* is supplied.

Input Keywords

Periodic — If present and nonzero, then the input data is periodic

Discussion

The function CORR1D computes the discrete correlation of two sequences *x* and *y*. If only one argument is passed, then CORR1D computes the discrete correlation of *x* and *x*.

More precisely, let *n* be the length of *x* and *y*. If *Periodic* is set, then $n_z = n$, otherwise n_z is set to the smallest whole number, $n_z \geq 2n - 1$, of the form

$$n_z = 2^{\alpha}3^{\beta}5^{\gamma} \quad : \alpha, \beta, \gamma \text{ nonnegative integers.}$$

The arrays *x* and *y* are then zero-padded to a length n_z. Then, we compute

$$z_i = \sum_{j=0}^{n_z - 1} x_{i+j} y_j$$

where the index on *x* is interpreted as a positive number between 0 and $n_z - 1$.

The technique used to compute the z_i's is based on the fact that the (complex discrete) Fourier transform maps correlation into multiplication. Thus, the Fourier transform of *z* is given by

$$\hat{z}_j = \hat{x}_j \overline{\hat{y}_j}$$

where the following equation is true.

$$\hat{z}_j = \sum_{m=0}^{n_z - 1} z_m e^{-2\pi i m n / n_z}$$

Thus, the technique used here to compute the correlation is to take the discrete Fourier transform of *x* and the conjugate of the discrete Fourier transform of *y*, multiply the results together component-wise, and then take the inverse transform of this product. It is very important to make sure that n_z is the product of small primes if *Periodic* is selected. If n_z is the product of small primes, then the computational effort will be proportional to $n_z \log(n_z)$. If *Periodic* is not set, then a good value is chosen for n_z so that the Fourier transforms are efficient and $n_z \geq 2n - 1$. This will mean that both vectors may be padded with zeros.

We point out that if x and y are not complex, then no complex transforms of x or y are taken, since a real transforms can simulate the complex transform above. Such a strategy is six times faster and requires less space than when using the complex transform.

Example

In this example, we compute a periodic correlation between two distinct signals x and y. We have 100 equally spaced points on the interval $[0, 2\pi]$ and $f_1(x) = \sin(x)$. We define x and y as follows:

$$x_i = f_1\left(\frac{2\pi i}{n-1}\right) \qquad i = 0, \ldots, n-1$$

$$y_i = f_1\left(\frac{2\pi i}{n-1} + \frac{\pi}{2}\right) \qquad i = 0, \ldots, n-1.$$

Note that the maximum value of z (the correlation of x with y) occurs at $i = 25$, which corresponds to the offset.

```
n = 100
t = 2*!DPI*FINDGEN(n)/(n-1)
x = SIN(t)
y = SIN(t+!dpi/2)
    ; Define the signals and compute the norms of the signals.
xnorm = NORM(x)
ynorm = NORM(y)
z = CORR1D(x, y, /Periodic)/(xnorm*ynorm)
    ; Compute the periodic correlation, and find the largest normalized
    ; element of the result.
max_z = (SORT(z))(N_ELEMENTS(z)-1)
PRINT, max_z, z(max_z)
   25   1.00
```

LAPLACE_INV Function

Computes the inverse Laplace transform of a complex function.

Usage

result = LAPLACE_INV(*f*, *sigma0*, *t*)

Input Parameters

***f* —** Scalar string specifying the user-supplied function for which the inverse Laplace transform will be computed.

***sigma0* —** An estimate for the maximum of the real parts of the singularities of *f*. If unknown, set *sigma0* = 0.0.

***t* —** One-dimensional array of size n containing the points at which the inverse Laplace transform is desired.

Returned Value

***result* —** One-dimensional array of length n whose *i*-th component contains the approximate value of the inverse Laplace transform at the point *t*(*i*).

Input Keywords

***Double* —** If present and nonzero, double precision is used.

***Pseudo_Acc* —** The required absolute uniform pseudo accuracy for the coefficients and inverse Laplace transform values.

Default: *Pseudo_Acc* = SQRT(ε), where ε is machine epsilon

***Sigma* —** The first parameter of the Laguerre expansion. If *Sigma* is not greater than *sigma0*, it is reset to *sigma0*+ 0.7.

Default: *Sigma* = *sigma0*+ 0.7

***Bvalue* —** The second parameter of the Laguerre expansion. If *Bvalue* is less than 2.0*(*Sigma* – *sigma0*), it is reset to 2.5*(*Sigma* – *sigma0*).

Default: *Bvalue* = 2.5*(*Sigma* – *sigma0*)

***Mtop* —** An upper limit on the number of coefficients to be computed in the Laguerre expansion. Keyword *Mtop* must be a multiple of four.

Default: *Mtop* = 1024

Output Keywords

***Err_Est* —** Named variable into which an overall estimate of the pseudo error, *Disc_Est* + *Trunc_Err* + *Cond_Err* is stored. See the *Discussion* section for details.

***Disc_Err* —** Named variable into which the estimate of the pseudo discretization error is stored.

***Trunc_Err* —** Named variable into which the estimate of the pseudo truncation error is stored.

***Cond_Err* —** Named variable into which the estimate of the pseudo condition error on the basis of minimal noise levels in the function values is stored.

***K* —** Named variable into which the coefficient of the decay function is stored. See the *Discussion* section for details.

***R* —** Named variable into which the base of the decay function is stored. See the *Discussion* section for details.

***Big_Coef_Log* —** Named variable into which the logarithm of the largest coefficient in the decay function is stored. See the *Discussion* section for details.

***Small_Coef_Log* —** Named variable into which the logarithm of the smallest nonzero coefficient in the decay function is stored. See the *Discussion* section for details.

***Indicators* —** Named variable into which an one-dimensional array of length n containing the overflow/underflow indicators for the computed approximate inverse Laplace transform is stored. For the i-th point at which the transform is computed, *Indicators*(i) signifies the following:

Indicators(i)	meaning
1	Normal termination.
2	The value of the inverse Laplace transform is too large to be representable. This component of the result is set to NaN.

Indicators(i)	meaning
3	The value of the inverse Laplace transform is found to be too small to be representable. This component of the result is set to 0.0.
4	The value of the inverse Laplace transform is estimated to be too large, even before the series expansion, to be representable. This component of the result is set to NaN.
5	The value of the inverse Laplace transform is estimated to be too small, even before the series expansion, to be representable. This component of the result is set to 0.0.

Discussion

The function LAPLACE_INV computes the inverse Laplace transform of a complex-valued function. Recall that if *f* is a function that vanishes on the negative real axis, then the Laplace transform of *f* is defined by

$$L[f](s) = \int_0^\infty e^{-sx} f(x)\,dx$$

It is assumed that for some value of *s* the integrand is absolutely integrable.

The computation of the inverse Laplace transform is based on a modification of Weeks' method (see Weeks (1966)) due to Garbow et al. (1988). This method is suitable when *f* has continuous derivatives of all orders on [0, ∞). In particular, given a complex-valued function $F(s) = L[f](s)$, *f* can be expanded in a Laguerre series whose coefficients are determined by *F*. This is fully described in Garbow et al. (1988) and Lyness and Giunta (1986).

The algorithm attempts to return approximations $g(t)$ to $f(t)$ satisfying

$$\left| \frac{g(t) - f(t)}{e^{\sigma t}} \right| < \varepsilon$$

where ε = *Pseudo_Acc* and σ = *Sigma* > *sigma0*. The expression on the left is called the pseudo error. An estimate of the pseudo error is available in *Err_Est*.

The first step in the method is to transform *F* to ϕ where

$$\phi(z) = \frac{b}{1-z} F(\frac{b}{1-z} - \frac{b}{2} + \sigma)$$

Then, if *f* is smooth, it is known that ϕ is analytic in the unit disc of the complex plane and hence has a Taylor series expansion

$$\phi(z) = \sum_{s=0}^{\infty} a_s z^s$$

which converges for all *z* whose absolute value is less than the radius of convergence R_c. This number is estimated in the output keyword *R*. Using the output keyword *K*, the smallest number *K* is estimated which satisfies

$$|a_s| < \frac{K}{R^s}$$

for all $R < R_c$.

The coefficients of the Taylor series for ϕ can be used to expand *f* in a Laguerre series

$$f(t) = e^{\sigma t} \sum_{s=0}^{\infty} a_s e^{-bt/2} L_s(bt)$$

Example 1

This example computes the inverse Laplace transform of the function $(s - 1)^{-2}$, and prints the computed approximation, true transform value, and difference at five points. The correct inverse transform is xe^x. From Abramowitz and Stegun (1964).

```
FUNCTION fcn,    x
   ; Return 1/(s - 1)**2
```

```
    one   =   COMPLEX(1.0,   0.0)
    f   =   one/((x - one)*(x - 1))
    RETURN, f
END

n   =   5
    ; Initialize t and compute inverse.
t   =   FINDGEN(n) + 0.5
l_inverse   =   LAPLACE_INV('fcn',   1.5,   t)
    ; Compute true inverse, relative difference.
true_inverse = t*EXP(t)
relative_diff = ABS((l_inverse - true_inverse) / true_inverse)
PM, [[t(0:*)],   [l_inverse(0:*)],   [true_inverse(0:*)],   $
     [relative_diff(0:*)]],   $
     Title   =   "           t            f_inv          true
   diff"
```

t	f_inv	true	diff
0.500000	0.824348	0.824361	1.48223e-05
1.50000	6.72247	6.72253	1.01432e-05
2.50000	30.4562	30.4562	2.50504e-07
3.50000	115.906	115.904	1.84310e-05
4.50000	405.053	405.077	5.90648e-05

Example 2

This example computes the inverse Laplace transform of the function $e^{-1/s}/s$, and prints the computed approximation, true transform value, and difference at five points. Additionally, the inverse is returned, and a required accuracy for the inverse transform values is specified. The correct inverse transform is

$$J_0\left(2\sqrt{x}\right)$$

```
FUNCTION fcn, x
    ; Return (1/s)(exp(-1/s)
```

```
    one   =   COMPLEX(1.0,   0.0)
    s_inverse = one / x
    f   =   s_inverse*EXP(-1*(s_inverse))
    RETURN, f
END

n   =   5
    ; Initialize t and compute inverse.
t   =   FINDGEN(n) + 0.5
l_inverse   =   LAPLACE_INV('fcn', 0.0, t, $
              Pseudo_Acc = 1.0e-6, Indicator = indicator)
    ; Compute true inverse, relative difference.
true_inverse = FLOAT(BESSJ(0, 2.0*SQRT(t)))
relative_diff = ABS((l_inverse - true_inverse) / true_inverse)
FOR i   =   0, 4 DO   $
    IF (indicator(i) EQ 0) THEN   $
         PM, t(i), l_inverse(i), true_inverse(i), $
              relative_diff(i), $
   Title   =   '          t            f_inv          true
    diff'   $
    ELSE   $
         PRINT, 'Overflow or underflow noted.'
          t            f_inv          true         diff
       0.500000      0.559134      0.559134   1.06602e-07
          t            f_inv          true         diff
       1.50000    -0.0229669    -0.0229670   4.21725e-06
          t            f_inv          true         diff
       2.50000     -0.310045     -0.310045   9.61226e-08
          t            f_inv          true         diff
       3.50000     -0.401115     -0.401115   2.22896e-07
          t            f_inv          true         diff
       4.50000     -0.370335     -0.370336   4.02369e-07
```

7

CHAPTER

Nonlinear Equations

Contents of Chapter

Introduction

Zeros of a Polynomial

A polynomial function of degree n can be expressed as follows:

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

where $a_n \neq 0$. Function ZEROPOLY finds zeros of a polynomial with real or complex coefficients using either the companion method or the Jenkins-Traub three-stage algorithm.

Zeros of a Function

Function ZEROFCN uses Müller's method to find the real zeros of a real-valued function.

Root of System of Equations

A system of equations can be stated as follows:

$f_i(x) = 0$, for $i = 0, 1, ..., n - 1$

where $x \in R^n$, and $f_i : R^n \rightarrow R$.

Function ZEROSYS uses a modified hybrid method due to M.J.D. Powell to find the zero of a system of nonlinear equations.

ZEROPOLY Function

Finds the zeros of a polynomial with real or complex coefficients using the companion matrix method or, optionally, the Jenkins-Traub, three-stage algorithm.

Usage

result = ZEROPOLY(*coef*)

Input Parameters

coef — Array containing the coefficients of the polynomial in increasing order by degree. The polynomial is $coef\,(n)\,z^n + coef\,(n-1)\,z^{n-1} + \ldots + coef\,(0)$.

Returned Value

result —The complex array of zeros of the polynomial.

Input Keywords

Double — If present and nonzero, double precision is used.

Companion — If present and nonzero, the companion matrix method is used.

Default: companion matrix method

Jenkins_Traub — If present and nonzero, the Jenkins-Traub, three-stage algorithm is used.

Discussion

Function ZEROPOLY computes the n zeros of the polynomial

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

where the coefficients a_i for $i = 0, 1, \ldots, n$ are real and n is the degree of the polynomial.

The default method used by ZEROPOLY is the companion matrix method. The companion matrix method is based on the fact that if C_a denotes the companion matrix associated with $p(z)$, then $\det(zI - C_a) = a(z)$, where I is an $n \times n$ identity matrix. Thus, $\det(z_0 I - C_a) = 0$ if, and only if, z_0 is a zero of $p(z)$. This implies that computing the eigenvalues of C_a will yield the zeros of $p(z)$. This method is thought to be more robust than the Jenkins-Traub algorithm in most cases, but the companion matrix method is not as computationally efficient. Thus, if speed is a concern, the Jenkins-Traub algorithm should be considered.

If the keyword *Jenkins_Traub* is set, then ZEROPOLY function uses the Jenkins-Traub three-stage algorithm (Jenkins and Traub 1970, Jenkins 1975). The zeros are computed one-at-a-time for real zeros or two-at-a-time for a complex conjugate pair. As the zeros are found, the real zero or quadratic factor is removed by polynomial deflation.

Example

This example finds the zeros of the third-degree polynomial

$$p(z) = z^3 - 3z^2 + 4z - 2$$

where z is a complex variable.

```
coef = [-2, 4, -3, 1]
    ; Set the coefficients.
zeros = ZEROPOLY(coef)
    ; Compute the zeros.
PM, zeros, Title = $
   'The complex zeros found are: '
       ; Print results.
The complex zeros found are:
(      1.00000,      0.00000)
```

```
(      1.00000,     -1.00000)
(      1.00000,      1.00000)
```

Warning Errors

MATH_ZERO_COEFF — First several coefficients of the polynomial are equal to zero. Several of the last roots are set to machine infinity to compensate for this problem.

MATH_FEWER_ZEROS_FOUND — Fewer than (N_ELEMENTS (*coef*) – 1) zeros were found. The root vector contains the value for machine infinity in the locations that do not contain zeros.

ZEROFCN Function

Finds the real zeros of a real function using Müller's method.

Usage

result = ZEROFCN(*f*)

Input Parameters

f — Scalar string specifying a user-supplied function for which the zeros are to be found. The *f* function accepts one scalar parameter from which the function is evaluated and returns a scalar of the same type.

Returned Value

result — An array containing the zeros *x* of the function.

Input Keywords

Double — If present and nonzero, double precision is used.

N_Roots — Number of roots to be found by ZEROFCN.

Default: *N_Roots* = 1

XGuess — Array with *N_Roots* components containing the initial guesses for the zeros.

Default: *XGuess* = 0

Err_Abs — First stopping criterion. A zero, x_i, is accepted if $| f(x_i) | < Err\_Abs$.

Default: *Err_Abs* = SQRT(ε), where ε is the machine precision

Err_Rel — Second stopping criterion. A zero, x_i, is accepted if the relative change of two successive approximations to x_i is less than *Err_Rel*.

Default: *Err_Rel* = SQRT(ε), where ε is the machine precision

Eta — Spread criteria for multiple zeros. If the zero, x_i, has been computed and $| x_i - x_j | < Eps$, where x_j is a previously computed zero, then the computation is restarted with a guess equal to $x_i + Eta$.

Default: *Eta* = 0.01

Eps — See *Eta*.

Default: *Eps* = SQRT(ε), where ε is the machine precision

Itmax — Maximum allowable number of iterations per zero.

Default: *Itmax* = 100

Info — Array of length *N_Roots* containing convergence information. The value *Info* (j – 1) is the number of iterations used in finding the j-th zero when convergence is achieved. If convergence is not obtained in *Itmax* iterations, *Info* (j – 1) is greater than *Itmax*.

Discussion

Function ZEROFCN computes n real zeros of a real function f. Given a user-supplied function $f(x)$ and an n-vector of initial guesses $x_0, x_1, \ldots, x_{n-1}$, the function uses Müller's method to locate n real zeros of f. The function has two convergence criteria. The first criterion requires that $| f(x_i^{(m)}) |$ be less than *Err_Abs*. The second criterion requires that the relative change of any two successive approximations to an x_i be less than *Err_Rel*. Here, $x_i^{(m)}$ is the m-th approximation to x_i. Let *Err_Abs* be denoted by ε_1, and *Err_Rel* be denoted by ε_2. The criteria can be stated mathematically as follows:

ZEROFCN has two convergence criteria; "convergence" is the satisfaction of either criterion.

Criterion 1:

$$\left| f(x_i^{(m)}) \right| < \varepsilon_1$$

Criterion 2:

$$\left| \frac{x_i^{(m+1)} - x_i^{(m)}}{x_i^{(m)}} \right| < \varepsilon_2$$

"Convergence" is the satisfaction of either criterion.

Example

This example finds a real zero of the third-degree polynomial:

$$f(x) = x^3 - 3x^2 + 3x - 1$$

```
.RUN
    ; Define function f.
- FUNCTION f, x
- return, x^3 - 3 * x^2 + 3 * x - 1
- END
!P.Font = 0
    ; Use hardware characters for the plot.
zero = ZEROFCN("f")
    ; Compute the real zero(s).
x = 2 * FINDGEN(100)/99
PLOT, x, f(x)
    ; Plot results.
OPLOT, [zero], [f(zero)], Psym = 6
XYOUTS, .5, .5, $
   'Computed zero is at x = ' + $
   STRING(zero(0)), Charsize = 1.5
```

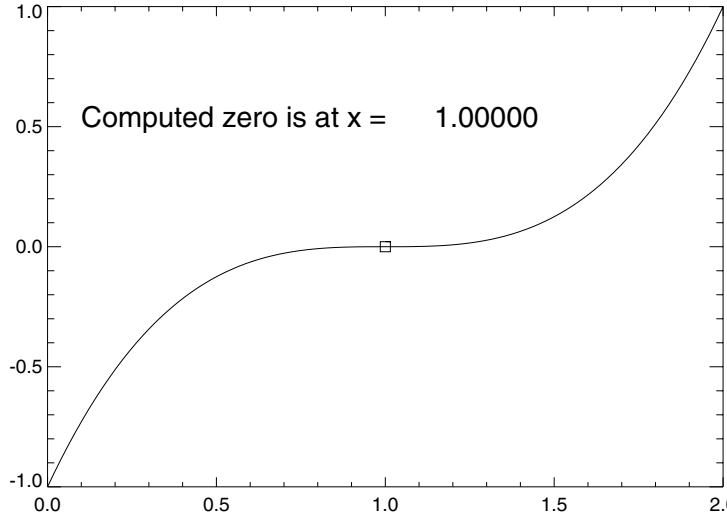


Figure 7-1 The ZEROFCN function finds the real zero of a third-degree polynomial.

Warning Errors

`MATH_NO_CONVERGE_MAX_ITER` — Function failed to converge within *Itmax* iterations for at least one of the *N_Roots* roots.

ZEROSYS Function

Solves a system of n nonlinear equations, $f_i(x) = 0$, using a modified Powell hybrid algorithm.

Usage

result = ZEROSYS(*f, n*)

Input Parameters

f — Scalar string specifying a user-supplied function to evaluate the system of equations to be solved. The *f* function accepts one parameter containing the point at which the functions are to be evaluated and returns the computed function values at the given point.

n — Number of equations to be solved and the number of unknowns.

Returned Value

result —An array containing a solution of the system of equations.

Input Keywords

Double — If present and nonzero, double precision is used.

Err_Rel — Stopping criterion. The root is accepted if the relative error between two successive approximations to this root is less than *Err_Rel*.

Default: *Err_Rel* = SQRT(ε), where ε is the machine precision

Jacobian — Scalar string specifying a user-supplied function to evaluate the x *n* Jacobian. The function accepts as parameter the point at which the Jacobian is to be evaluated and returns a two-dimensional matrix defined by result (*i, j*) = $\partial f_i / \partial x_j$.

Itmax — Maximum allowable number of iterations.

Default: *Itmax* = 200

XGuess — Array with *N* components containing the initial estimate of the root.

Default: *XGuess* = 0

Output Keywords

Fnorm — Scalar with the value $f^2_0 + \ldots + f^2_{n-1}$ at the point *x*.

Discussion

Function ZEROSYS is based on the MINPACK subroutine HYBRDJ, which uses a modification of the hybrid algorithm due to M.J.D. Powell. This algorithm is a variation of Newton's Method, which takes precautions to avoid undesirable large steps or increasing residuals. For further discussion, see Moré et al. (1980).

Example

The following 2 x 2 system of nonlinear equations is solved:

$$f(x) = x_0 + x_1 - 3$$

$$f(x) = x_0^2 + x_1^2 - 9$$

```
.RUN
    ; Define the system through the function f.
- FUNCTION f, x
- RETURN, [x(0)+x(1)-3, x(0)^2+x(1)^2-9]
- END
PM, ZEROSYS("f", 2), $
   Title = 'Solution of the system:', $
   Format = '(f10.5)'
       ; Compute the solution and output the results.
Solution of the system:
   0.00000
   3.00000
```

Warning Errors

MATH_TOO_MANY_FCN_EVALS — Number of function evaluations has exceeded *Itmax*. A new initial guess can be tried.

MATH_NO_BETTER_POINT — Keyword *Err_Rel* is too small. No further improvement in the approximate solution is possible.

MATH_NO_PROGRESS — Iteration has not made good progress. A new initial guess can be tried.

8

CHAPTER

Optimization

Contents of Chapter

Introduction

Unconstrained Minimization

The unconstrained minimization problem can be stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

where $f : R^n \rightarrow R$

is continuous and has derivatives of all orders required by the algorithms. The functions for unconstrained minimization are grouped into three categories: univariate functions, multivariate functions, and nonlinear least-squares functions.

For the univariate functions, it is assumed that the function is unimodal within the specified interval. For discussion on unimodality, see Brent (1973).

A quasi-Newton method is used for the multivariate function FMINV. The default is to use a finite-difference approximation of the gradient of $f(x)$. Here, the gradient is defined to be the following vector:

$$\nabla f(x) = \left[\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots, \frac{\partial f(x)}{\partial x_n}\right]$$

However, when the exact gradient can be easily provided, the *Grad* keyword should be used.

The nonlinear least-squares function uses a modified Levenberg-Marquardt algorithm. The most common application of the function is the nonlinear data-fitting problem where the user is trying to fit the data with a nonlinear model.

These functions are designed to find only a local minimum point. However, a function may have many local minima. Try different initial points and intervals to obtain a better local solution.

Double-precision arithmetic is recommended for the functions when the user provides only the function values.

Linearly Constrained Minimization

The linearly constrained minimization problem can be stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

subject to

$$A_1 x = b_1$$

$$A_2 x \geq b_2$$

where $f : R^n \rightarrow R$, A_1 and A_2

are coefficient matrices and b_1 and b_2 are vectors. If $f(x)$ is linear, then the problem is a linear programming problem; if $f(x)$ is quadratic, the problem is a quadratic programming problem.

Function LINPROG uses a revised simplex method to solve small- to medium-sized linear programming problems. No sparsity is assumed since the coefficients are stored in full matrix form.

The QUADPROG function is designed to solve convex quadratic programming problems using a dual quadratic programming algorithm. If the given Hessian is not positive definite, then QUADPROG modifies it to be positive definite. In this case, output should be interpreted with care because the problem has been changed slightly. Here, the Hessian of $f(x)$ is defined to be the $n \times n$ matrix as follows:

$$\nabla^2 f(x) = \left[\frac{\partial^2}{\partial x_i \partial x_j} f(x) \right]$$

Nonlinearly Constrained Minimization

The nonlinearly constrained minimization problem can be stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

subject to

$$g_i(x) = 0 \qquad \text{for} \quad i = 1, 2, \ldots, m_1$$

$$g_i(x) \geq 0 \qquad \text{for} \quad i = m_1 + 1, \ldots, m$$

where $f : R^n \rightarrow R$ and $g_i : R^n \rightarrow R$ for $i = 1, 2, ..., m$.

Function NONLINPROG uses a successive quadratic programming algorithm to solve this problem. A more complete discussion of this algorithm can be found on page 338.

FMIN Function

Finds the minimum point of a smooth function $f(x)$ of a single variable using function evaluations and, optionally, through both function evaluations and first derivative evaluations.

Usage

result = FMIN(*f, a, b* [, *grad*])

Input Parameters

f — Scalar string specifying a user-supplied function to compute the value of the function to be minimized. Parameter *f* accepts the following parameter and returns the computed function value at this point:

x — Point at which the function is to be evaluated.

a — Lower endpoint of the interval in which the minimum point of *f* is to be located.

b — Upper endpoint of the interval in which the minimum point of *f* is to be located.

grad — Scalar string specifying a user-supplied function to compute the first derivative of the function. Parameter *grad* accepts the following parameter and returns the computed derivative at this point:

x — Point at which the derivative is to be evaluated.

Returned Value

result —The point at which a minimum value of *f* is found. If no value can be computed, then NaN (Not a Number) is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

XGuess — Initial guess of the minimum point of *f*.

Default: *XGuess* = (*a* + *b*) / 2

Max_Evals — Maximum number of function evaluations allowed.

Default: *Max_Evals* = 1000

Err_Abs — Required absolute accuracy in the final value of *x*. On a normal return, there are points on either side of *x* within a distance *Err_Abs* at which *f* is no less than *f* at *x*. Keyword *Err_Abs* cannot be used if the optional parameter *grad* is supplied.

Default: *Err_Abs* = 0.0001

Step — Order of magnitude estimate of the required change in *x*. Keyword *Step* cannot be used if the optional parameter *grad* is supplied.

Default: *Step* = 1.0

Err_Rel — Required relative accuracy in the final value of *x*. This is the first stopping criterion. On a normal return, the solution *x* is in an interval that contains a local minimum and is less than or equal to max (1.0, | *x* |) * *Err_Rel*. When the given *Err_Rel* is less than zero, SQRT(ε) is used as *Err_Rel*, where ε is the machine precision. Keyword *Err_Rel* can only be used if the optional parameter *grad* is supplied.

Default: *Err_Rel* = SQRT(ε)

Tol_Grad — Derivative tolerance used to decide if the current point is a local minimum. This is the second stopping criterion. Parameter *x* is returned as a solution when *grad* is less than or equal to *Tol_Grad*. Keyword *Tol_Grad* should be nonnegative; otherwise, zero is used. Keyword *Tol_Grad* can only be used if the optional parameter *grad* is supplied.

Default: *Tol_Grad* = SQRT(ε), where ε is the machine precision

Output Keywords

FValue — Function value at point *x*. Keyword *FValue* can only be used if the optional parameter *grad* is supplied.

GValue — Derivative value at point *x*. Keyword *GValue* can only be used if the optional parameter *grad* is supplied.

Discussion

Function FMIN uses a safeguarded, quadratic interpolation method to find a minimum point of a univariate function. Both the code and the underlying algorithm are based on the subroutine ZXLSF written by M.J.D. Powell at the University of Cambridge.

The FMIN function finds the least value of a univariate function, *f*, which is specified by the function *f*. (Other required data are two points *A* and *B* that define an interval for finding a minimum point from an initial estimate of the solution, x_0, where $x_0 = XGuess$.) The algorithm begins the search by moving from x_0 to $x = x_0 + s$, where $s = Step$ is an estimate of the required change in x and may be positive or negative. The first two function evaluations indicate the direction to the minimum point, and the search strides out along this direction until a bracket on a minimum point is found or until x reaches one of the endpoints *a* or *b*. During this stage, the step length increases by a factor of between 2 and 9 per function evaluation. The factor depends on the position of the minimum point that is predicted by quadratic interpolation of the three most recent function values.

When an interval containing a solution has been found, the three points are as follows:

$$x_1, x_2, x_3, \text{ with } x_1 < x_2 < x_3, \quad f(x_1) \geq f(x_2), \text{ and } f(x_2) \geq f(x_3)$$

The following rules should be considered when choosing the new x from these three points:

- the estimate of the minimum point that is given by quadratic interpolation of the three function values
- a tolerance parameter η, which depends on the closeness of $|f|$ to a quadratic
- whether x_2 is near the center of the range between x_1 and x_3 or is relatively close to an end of this range

In outline, the new value of x is as near as possible to the predicted minimum point, subject to being at least ε from x_2 and subject to being in the longer interval between x_1 and x_2 or x_2 and x_3, when x_2 is particularly close to x_1 or x_3.

The algorithm is intended to provide fast convergence when f has a positive and continuous second derivative at the minimum and to avoid gross inefficiencies in pathological cases, such as the following:

$$f(x) = x + 1.001 \mid x \mid$$

The algorithm can make ε large automatically in the pathological cases. In this case, it is usual for a new value of x to be at the midpoint of the longer interval that is adjacent to the least calculated function value. The midpoint strategy is used frequently when changes to f are dominated by computer rounding errors, which happens if the user requests an accuracy that is less than the square root of the machine precision. In such cases, the subroutine claims to have achieved the required accuracy if it decides that there is a local minimum point within distance δ of x, where δ = *Err_Abs*, even though the rounding errors in f may cause the existence of other local minimum points nearby. This difficulty is inevitable in minimization routines that use only function values, so high-precision arithmetic is recommended.

If parameter *grad* is supplied, then the FMIN function uses a descent method with either the secant method or cubic interpolation to find a minimum point of a univariate function. It starts with an initial guess and two endpoints. If any of the three points is a local minimum point and has least function value, the function terminates with a solution; otherwise, the point with least function value is used as the starting point.

From the starting point, for example x_c, the function value $f_c = f(x_c)$, the derivative value $g_c = g(x_c)$, and a new point x_n, defined by $x_n = x_c - g_c$, are computed. The function $f_n = f(x_n)$ and the derivative $g_n = g(x_n)$ are then evaluated. If either
$f_n \geq f_c$ or g_n has the opposite sign of g_c, then a minimum point exists between x_c and x_n, and an initial interval is obtained; otherwise, since x_c is kept as the point that has lowest function value, an interchange between x_n and x_c is performed. The secant method is then used to get a new point:

$$x_s = x_c - g_c\left(\frac{g_n - g_c}{x_n - x_c}\right)$$

Let $x_n \leftarrow x_s$. Repeat this process until an interval containing a minimum is found or one of the following convergence criteria is satisfied:

Criterion 1: $| x_c - x_n | \leq \varepsilon_c$

Criterion 2: $| g_c | \leq \varepsilon_g$

where $\varepsilon_c = \max \{1.0, | x_c |\} * \varepsilon$, ε is a relative error tolerance and ε_g is a gradient tolerance.

When convergence is not achieved, a cubic interpolation is performed to obtain a new point. The function and derivative are then evaluated at that point; accordingly, a smaller interval that contains a minimum point is chosen. A safeguarded method is used to ensure that the interval be reduced by at least a fraction of the previous interval. Another cubic interpolation is then performed, and this function is repeated until one of the stopping criteria is met.

Example 1

A minimum point of $f(x) = e^x - 5x$ is found.

```
.RUN
    ; Define the function to be used.
- FUNCTION f, x
- RETURN, EXP(x) - 5 * x
- END
xmin = FMIN('f', -100, 100)
    ; Call FMIN to compute the minimum.
PM, xmin
    ; Print results.
       1.60943
x = 10 * FINDGEN(100)/99 - 5
!P.Font = 0
PLOT, x, f(x), $
   Title = '!8f(x) = e!Ex!N-5x!3', $
   XTitle = 'x', YTitle = 'f(x)'
       ; Plot results.
OPLOT, [xmin], [f(xmin)], Psym = 6
str = '(' + STRCOMPRESS(xmin) + ',' + $
   STRCOMPRESS(f(xmin)) + ')'
OPLOT, [xmin],[f(xmin)], Psym = 6
XYOUTS, -5, 80, 'Minimum point:!C' + str, $
   Charsize = 1.2
```

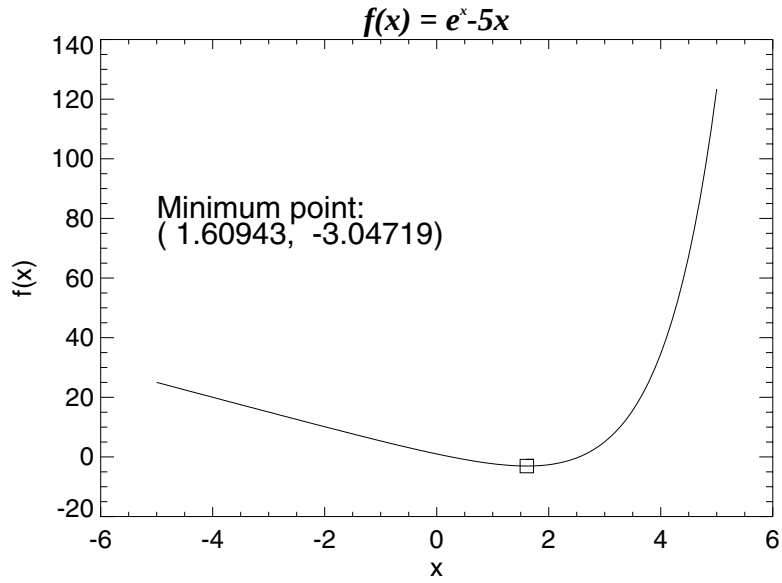


Figure 8-1 Minimum point of a smooth function.

Example 2

In this example, parameter *grad* is supplied, and a minimum point of $f(x) = x\,(x^3 - 1\,) + 10$ is found with an initial guess $x_0 = 3$.

```
.RUN
- FUNCTION f, x
- RETURN, x * (x^3 - 1) + 10
- END

.RUN
- FUNCTION grad, x
- RETURN, 4 * x^3 - 1
- END

xmin = FMIN('f', -10, 10, 'grad')
x = 4 * FINDGEN(100)/99 - 2

!P.Font = 0
PLOT, x, f(x), $
   Title = '!8f(x) = x(x!E3!N-1)+10!3', $
   XTitle ='x', YTitle = 'f(x)'

OPLOT, [xmin], [f(xmin)], Psym = 6
```

```
str = '(' + STRCOMPRESS(xmin) + ',' + STRCOMPRESS(f(xmin)) + ')'
XYOUTS, -1.5, 25, 'Minimum point:'+str, Charsize = 1.2
```

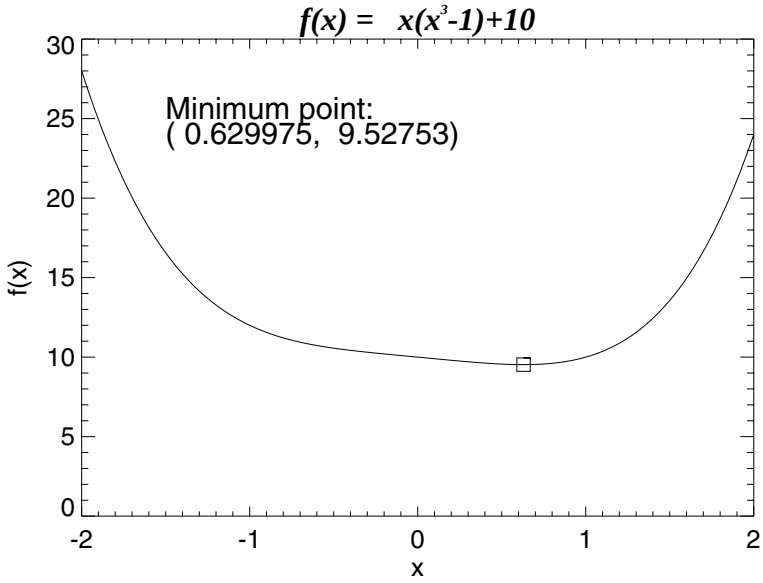


Figure 8-2 Minimum point of a smooth function.

Warning Errors

MATH_MIN_AT_LOWERBOUND — Final value of *x* is at the lower bound.

MATH_MIN_AT_UPPERBOUND — Final value of *x* is at the upper bound.

MATH_MIN_AT_BOUND — Final value of *x* is at a bound.

MATH_NO_MORE_PROGRESS — Computer rounding errors prevent further refinement of *x*.

MATH_TOO_MANY_FCN_EVAL — Maximum number of function evaluations exceeded.

FMINV Function

Minimizes a function *f* (*x*) of *n* variables using a quasi-Newton method.

Usage

result = FMINV(*f*, *n*)

Input Parameters

f — Scalar string specifying a user-supplied function to evaluate the function to be minimized. The *f* function accepts the following parameter and returns the computed function value at the point:

x — Point at which the function is evaluated.

n — Number of variables.

Returned Value

result — The minimum point *x* of the function. If no value can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

XGuess — Array with *n* components containing an initial guess of the computed solution.

Default: *XGuess* (*) = 0

Grad — Scalar string specifying a user-supplied function to compute the gradient. This function accepts the following parameter and returns the computed gradient at the point:

x — Point at which the gradient is evaluated.

XScale — Array with *n* components containing the scaling vector for the variables. Keyword *XScale* is used mainly in scaling the gradient and the distance between two points. See keywords *Tol_Grad* and *Tol_Step* for more detail.

Default: *XScale* (*) = 1.0

FScale — Scalar containing the function scaling. Keyword *Fscale* is used mainly in scaling the gradient. See keyword *Tol_Grad* for more detail.

Default: *FScale* = 1.0

Tol_Grad — Scaled gradient tolerance.

The *i*-th component of the scaled gradient at *x* is calculated as

$$\frac{|g_i| \times \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)}$$

where

$g = \nabla f(x)$,

s = *XScale*, and f_s = *FScale*.

Default: *Tol_Grad* = $\varepsilon^{1/2}$ ($\varepsilon^{1/3}$ in double) where ε is the machine precision

Tol_Step — Scaled step tolerance.

The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where s = *XScale*.

Default: *Tol_Step* = $\varepsilon^{2/3}$

Tol_Rfcn — Relative function tolerance.

Default: *Tol_Rfcn* = $\max(10^{-10}, \varepsilon^{2/3})$, $\max(10^{-20}, \varepsilon^{2/3})$ in double

Max_Step — Maximum allowable step size.

Default: *Max_Step* = $1000\max(\varepsilon_1, \varepsilon_2)$, where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} (s_i t_i)^2}$$

$\varepsilon_2 = \| s \|_2$, s = *XScale*, and t = *XGuess*

N_Digit — Number of good digits in the function.

Default: machine dependent

Itmax — Maximum number of iterations.

Default: *Itmax* = 100

Max_Evals — Maximum number of function evaluations.

Default: *Max_Evals* = 400

Max_Grad — Maximum number of gradient evaluations.

Default: *Max_Grad* = 400

Ihess — Hessian initialization parameter. If *Ihess* is zero, the Hessian is initialized to the identity matrix; otherwise, it is initialized to a diagonal matrix containing max ($f(t)$, f_s) * s_i on the diagonal, where t = *XGuess*, f_s = *FScale*, and s = *XScale*.

Default: *Ihess* = 0

Output Keywords

FValue — Name of a variable into which the value of the function at the computed solution is stored.

Discussion

Function FMINV uses a quasi-Newton method to find the minimum of a function $f(x)$ of n variables. The problem is stated below.

$$\min_{x \in \mathbf{R}^n} f(x)$$

Given a starting point x_c, the search direction is computed according to the formula

$$d = -B^{-1}g_c$$

where B is a positive definite approximation of the Hessian and g_c is the gradient evaluated at x_c.

A line search is then used to find a new point

$$x_n = x_c + \lambda d,\ \lambda > 0$$

such that

$$f(x_n) \leq f(x_c)\ \alpha g^T d$$

where $\alpha \in (0, 0.5)$.

Finally, the optimality condition

$$\| g(x) \| \leq \varepsilon$$

is checked, where ε is a gradient tolerance.

When optimality is not achieved, B is updated according to the BFGS formula

$$B \leftarrow B - \frac{Bss^T B}{s^T Bs} + \frac{yy^T}{y^T s}$$

where $s = x_n - x_c$ and $y = g_n - g_c$. Another search direction is then computed to begin the next iteration. For more details, see Dennis and Schnabel (1983, Appendix A).

In this implementation, the first stopping criterion for FMINV occurs when the norm of the gradient is less than the given gradient tolerance *Tol_Grad.* The second stopping criterion for FMINV occurs when the scaled distance between the last two steps is less than the step tolerance *Tol_Step*.

Since by default, a finite-difference method is used to estimate the gradient for some single-precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high-precision arithmetic is recommended or keyword *Grad* is used to provide more accurate gradient evaluation.

Example 1

The function $f(x) = 100\ (\ x_2 - x_1^2)^{\ 2} + (\ 1 - x_1)^2$ is minimized.

```
.RUN
    ; Define the function.
- FUNCTION f, x
- xn = x
- xn(0) = x(1) - x(0)^2
- xn(1) = 1 - x(0)
- RETURN, 100 * xn(0)^2 + xn(1)^2
- END
xmin = FMINV("f", 2)
    ; Call FMINV to compute the minimum.
```

```
PM, xmin, Title = 'Solution:'
    ; Output the solution.
Solution:
     0.999986
     0.999971
PM, f(xmin), Title = 'Function value:'
Function value:
     2.09543e-10
```

Example 2

The function $f(x) = 100 (x_2 – x_1^2) ^2 + (1 – x_1)^2$ is minimized with the initial guess $x = (–1.2, 1.0)$. In the following plot, the asterisk marks the minimum.

```
.RUN
    ; Define the function.
- FUNCTION f, x
- xn = x
- xn(0) = x(1) - x(0)^2
- xn(1) = 1 - x(0)
- RETURN, 100 * xn(0)^2 + xn(1)^2
- END
.RUN
    ; Define the gradient function.
- FUNCTION grad, x
- g = x
- g(0) = -400 * (x(1) - x(0)^2) * x(0) - $
-    2 * (1 - x(0))
- g(1) = 200 * (x(1) - x(0)^2)
- RETURN, g
- END
xmin = FMINV('f', 2, grad = 'grad',$
   XGuess = [-1.2, 1.0], Tol_Grad = .0001)
       ; Call FMINV with the gradient function, an initial guess, and a
       ; scaled gradient tolerance.
x = 4 * FINDGEN(100)/99 - 2
y = x
surf = FLTARR(100, 100)
FOR i = 0, 99 DO FOR j = 0, 99 do $
   surf(i, j) = f([x(i), y(j)])
```

```
        ; Evaluate the function f on a 100 x 100 grid for use in
        ; CONTOUR.
str = '(' + STRCOMPRESS(xmin(0)) + ',' + $
    STRCOMPRESS(xmin(1)) + ',' + $
    STRCOMPRESS(f(xmin)) + ')'
!P.Charsize = 1.5
CONTOUR, surf, x, y, Levels = $
    [20*FINDGEN(6), 500 + FINDGEN(7)*500], $
    /C_Annotation, $
    Title = '!18Rosenbrock Function!C' + $
    'Minimum Point:!C' + str, $
    Position = [.1, .1, .8, .8]
        ; Call CONTOUR. Customize the contour plot, including the title
        ; of the plot.
OPLOT, [xmin(0)], [xmin(1)], Psym = 2, $
    Symsize = 2
        ; Plot the solution as an asterisk.
```

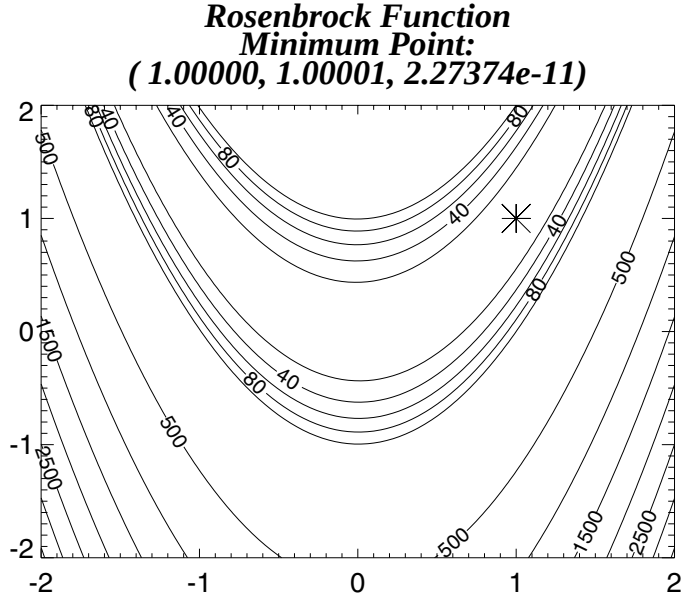


Figure 8-3 Plot of the Rosenbrock function.

Informational Errors

MATH_STEP_TOLERANCE — Scaled step tolerance satisfied. Current point may be an approximate local solution, but it is also possible that the algorithm is making very slow progress and is not near a solution or that *Tol_Step* is too big.

Warning Errors

MATH_REL_FCN_TOLERANCE — Relative function convergence. Both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance.

MATH_TOO_MANY_ITN — Maximum number of iterations exceeded.

MATH_TOO_MANY_FCN_EVAL — Maximum number of function evaluations exceeded.

MATH_TOO_MANY_GRAD_EVAL — Maximum number of gradient evaluations exceeded.

MATH_UNBOUNDED — Five consecutive steps have been taken with the maximum step length.

MATH_NO_FURTHER_PROGRESS — Last global step failed to locate a point lower than the current *x* value.

Fatal Errors

MATH_FALSE_CONVERGENCE — Iterates appear to be converging to a noncritical point. It is possible that either incorrect gradient information is used, or the function is discontinuous, or the other stopping tolerances are too tight.

NLINLSQ Function

Solves a nonlinear least-squares problem using a modified Levenberg-Marquardt algorithm.

Usage

result = NLINLSQ(*f, m, n*)

Input Parameters

f — Scalar string specifying a user-supplied function to evaluate the function that defines the least-squares problem. The *f* function accepts the following two parameters and returns an array of length *m* containing the function values at *x*:

m — Number of functions.

x — Array length *n* containing the point at which the function is evaluated.

m — Number of functions.

n — Number of variables where $n \leq m$.

Returned Value

result —The solution *x* of the nonlinear least-squares problem. If no solution can be computed, NULL is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

XGuess — Array with *n* components containing an initial guess.

Default: *XGuess* (*) = 0

Jacobian — Scalar string specifying a user-supplied function to compute the Jacobian. This function accepts two parameters and returns an *n* x *m* array containing the Jacobian at the point *s* input point. Note that each derivative $\partial f_i / \partial x_j$ should be returned in the (*i, j*) element of the returned matrix. The parameters of the function are as follows:

m Number of equations.

x Array of length *n* at which the point Jacobian is evaluated.

XScale — Array with *n* components containing the scaling vector for the variables. Keyword *XScale* is used mainly in scaling the gradient and the distance between two points. See keywords *Tol_Grad* and *Tol_Step* for more detail.

Default: *XScale* (*) = 1

FScale — Array with *m* components containing the diagonal scaling matrix for the functions. The *i*-th component of *FScale* is a positive scalar specifying the reciprocal magnitude of the *i*-th component function of the problem.

Default: *FScale* (*) = 1

Tol_Grad — Scaled gradient tolerance.

The *i*-th component of the scaled gradient at *x* is calculated as

$$\frac{|g_i| \times \max(|x_i|, 1/s_i)}{\frac{1}{2}\|F(x)\|_2^2}$$

where

$g = \nabla F(x)$, $s = XScale$, and $\|F(x)\|_2^2 = \sum_{i=1}^{m} f_i(x)^2$.

Default: *Tol_Grad* = $\varepsilon^{1/2}$ ($\varepsilon^{1/3}$ in double), where ε is the machine precision

Tol_Step — Scaled step tolerance.

The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_y|}{\max(|x_i|, 1/s_i)}$$

where *s* = *XScale*.

Default: *Tol_Step* = $\varepsilon^{2/3}$, where ε is the machine precision

Tol_Rfcn — Relative function tolerance.

Default: *Tol_Rfcn* = max(10^{-10}, $\varepsilon^{2/3}$), [max(10^{-40}, $\varepsilon^{2/3}$) in double], where ε is the machine precision

Tol_Afcn — Absolute function tolerance.

Default: *Tol_Afcn* = max(10^{-20}, ε^2), [max(10^{-40}, ε^2) in double], where ε is the machine precision

Max_Step — Maximum allowable step size.

Default: *Max_Step* = 1000max(ε_1, ε_2), where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} s_i t_i^2}$$

$$\varepsilon_2 = \|s\|_2,$$

s = *XScale*, and t = *XGuess*

Trust_Region — Size of initial trust-region radius.

Default: based on the initial scaled Cauchy step

N_Digits — Number of good digits in the function.

Default: machine dependent

Itmax — Maximum number of iterations.

Default: *Itmax* = 100

Max_Evals — Maximum number of function evaluations.

Default: *Max_Evals* = 400

Max_Jacobian — Maximum number of Jacobian evaluations.

Default: *Max_Jacobian* = 400

Intern_Scale — Internal variable scaling option. With this keyword, the values for *XScale* are set internally.

Tolerance — Tolerance used in determining linear dependence for the computation of the inverse of J^TJ. If *Jacobian* is specified, *Tolerance* = 100ε, where ε is the machine precision, is the default; otherwise, SQRT(ε), where ε is the machine precision, is the default.

Output Keywords

Fvec — Name of the variable into which a real array of length *m* containing the residuals at the approximate solution is stored.

Fjac — Name of the variable into which an array of size $m \times n$ containing the Jacobian at the approximate solution is stored.

Rank — Name of the variable into which the rank of the Jacobian is stored.

JTJ_inverse — Name of the variable into which an array of size $n \times n$ containing the inverse matrix of J^TJ, where J is the final Jacobian, is stored. If J^TJ is singular, the inverse is a symmetric g_2 inverse of J^TJ. (See CHNNDSOL on page 39 for a discussion of generalized inverses and the definition of the g_2 inverse.) See CHNNDSOL for a discussion of generalized inverses and the definition of the g_2 inverse.

Xlb — One dimensional array with n components containing the lower bounds on the variables. Keywords *Xlb* and *Xub* must be used together.

Xub — One dimensional array with n components containing the upper bounds on the variables. Keywords *Xlb* and *Xub* must be used together.

Discussion

The specific algorithm used in NLINLSQ is dependent on whether the keywords *Xlb* and *Xub* are supplied. If keywords *Xlb* and *Xub* are not supplied, then the function NLINLSQ is based on the MINPACK routine LMDER by Moré et al. (1980).

Function NLINLSQ, based on the MINPACK routine LMDER by Moré et al. (1980), uses a modified Levenberg-Marquardt method to solve nonlinear least-squares problems. The problem is stated as follows:

$$\min \frac{1}{2}F(x)^TF(x) = \frac{1}{2}\sum_{i=1}^{m} f_i(x)^2$$

where $m \geq n$, $F : R^n \rightarrow R^m$ and $f_i(x)$

is the i-th component function of $F(x)$. From a current point, the algorithm uses the trust region approach

$$\min_{x \in \mathbf{R}^n} \left\| F(x_c) + J(x_c)(x_n - x_c) \right\|_2$$

subject to $\left\| x_n - x_c \right\|_2 \leq \delta_c$

to get a new point x_n. Compute x_n as

$$x_n = x_c - (J(x_c)^T J(x_c) + \mu_c I)^{-1} J(x_c)^T F(x_c)$$

where $\mu_c = 0$ if $\delta_c \geq \| (J(x_c)^T J(x_c))^{-1} J(x_c)^T F(x_c) \|_2$ and $\mu_c > 0$ otherwise.

The value μ_c is defined by the function. The vector and matrix $F(x_c)$ and $J(x_c)$ are the function values and the Jacobian evaluated at the current point x_c. This function is repeated until the stopping criteria are satisfied.

The first stopping criterion for NLINLSQ occurs when the norm of the function is less than the absolute function tolerance, *Tol_Afcn*. The second stopping criterion occurs when the norm of the scaled gradient is less than the given gradient tolerance *Tol_Grad*. The third stopping criterion for NLINLSQ occurs when the scaled distance between the last two steps is less than the step tolerance *Tol_Step*. For more details, see Levenberg (1944), Marquardt (1963), or Dennis and Schnabel (1983, Chapter 10).

If keywords *Xlb* and *Xub* are supplied, then the function NLINLSQ uses a modified Levenberg-Marquardt method and an active set strategy to solve nonlinear least-squares problems subject to simple bounds on the variables. The problem is stated as follows:

$$\min \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2$$

subject to $l \leq x \leq u$ where $m \geq n$, $F : \mathbf{R}^n \rightarrow \boldsymbol{R}^m$, *and* $f_i(x)$ is the *i*-th component function of $F(x)$. From a given starting point, an active set IA, which contains the indices of the variables at their bounds, is built. A variable is called a "free variable" if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -(J^T J + \mu I)^{-1} J^T F$$

where μ is the Levenberg-Marquardt parameter, $F = F(x)$, and J is the Jacobian with respect to the free variables. The search direction for the variables in IA is set to zero. The trust region approach discussed by Dennis and Schnabel (1983) is used to find the new point. Finally, the optimality conditions are checked. The conditions are:

$$||g(x_i)|| \le \varepsilon,\ l_i < x_i < u_i$$

$$g(x_i) < 0,\ x_i = u_i$$

$$g(x_i) > 0,\ x_i = l_i$$

where ε is a gradient tolerance. This process is repeated until the optimality criterion is achieved.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in IA, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of IA. For more detail on the Levenberg-Marquardt method, see Levenberg (1944) or Marquardt (1963). For more detail on the active set strategy, see Gill and Murray (1976).

Since a finite-difference method is used to estimate the Jacobian for some single-precision calculations, an inaccurate estimate of the Jacobian may cause the algorithm to terminate at a noncritical point. In such cases, high-precision arithmetic is recommended. Also, whenever the exact Jacobian can be easily provided, the keyword *Jacobian* should be used.

Example

In this example, the nonlinear data-fitting problem found in Dennis and Schnabel (1983, p. 225),

$$\min \frac{1}{2} \sum^{3} f_i(x)^2 \quad \text{where } f_i(x) = \ f_i(x) \ = \ e^{t_i x} - y_i \ ,$$

is solved with the data t = [1, 2, 3] and y = [2, 4, 3].

```
.RUN
    ; Define the function that defines the least-squares problem.
- FUNCTION f, m, x
- y = [2, 4, 3]
- t = [1, 2, 3]
- RETURN, EXP(x(0) * t) - y
- END
solution = NLINLSQ("f", 3, 1)
    ; Call NLINLSQ.

PM, solution, Title = 'The solution is:'
```

```
    ; Output the results.
The solution is:
     0.440066
PM, f(m, solution), $
   Title = 'The function values are:'
The function values are:
    -0.447191
     -1.58878
     0.744159
```

Informational Errors

MATH_STEP_TOLERANCE — Scaled step tolerance satisfied. The current point may be an approximate local solution, but it is also possible that the algorithm is making very slow progress and is not near a solution or that *Tol_Step* is too big.

Warning Errors

MATH_LITTLE_FCN_CHANGE — Both the actual and predicted relative reductions in the function are less than or equal to the relative function tolerance.

MATH_TOO_MANY_ITN — Maximum number of iterations exceeded.

MATH_TOO_MANY_FCN_EVAL — Maximum number of function evaluations exceeded.

MATH_TOO_MANY_JACOBIAN_EVAL — Maximum number of Jacobian evaluations exceeded.

MATH_UNBOUNDED — Five consecutive steps have been taken with the maximum step length.

Fatal Errors

MATH_FALSE_CONVERGE — Iterates appear to be converging to a noncritical point.

LINPROG Function

Solves a linear programming problem using the revised simplex algorithm.

Usage

result = LINPROG(*a*, *b*, *c*)

Input Parameters

a — Two-dimensional matrix containing the coefficients of the constraints. The coefficient for the *i*-th constraint is contained in *A* (*i*, *).

b — One-dimensional matrix containing the right-hand side of the constraints. If there are limits on both sides of the constraints, *b* contains the lower limit of the constraints.

c — One-dimensional array containing the coefficients of the objective function.

Returned Value

result —The solution *x* of the linear programming problem.

Input Keywords

Double — If present and nonzero, double precision is used.

Bu — Array with N_ELEMENTS(*b*) elements containing the upper limit of the constraints that have both the lower and the upper bounds. If no such constraint exists, *Bu* is not needed.

Irtype — Array with N_ELEMENTS(*b*) elements indicating the types of general constraints in the matrix *A*. Let $r_i = A_{i0}x_0 + \ldots + A_{in-1}\,x_{n-1}$. The value of *Irtype* (*i*) is described in the table below.

Irtype (i)	Constraints
0	$r_i = b_i$
1	$r_i \le bu$
2	$r_i \ge b_i$
3	$b_i \le r_i \le bu$

Default: *Irtype* (*) = 0

Xlb — Array with N_ELEMENTS(*c*) elements containing the lower bound on the variables. If there is no lower bound on a variable, 10^{30} should be set as the lower bound.

Default: *Xlb* (*) = 0

Xub — Array with N_ELEMENTS(*c*) elements containing the upper bound on the variables. If there is no upper bound on a variable, -10^{30} should be set as the upper bound.

Default: *Xub* (*) = *infinity*

Itmax — Maximum number of iterations.

Default: *Itmax* = 10,000

Output Keywords

Obj — Name of the variable into which the optimal value of the objective function is stored.

Dual — Name of the variable into which the array with N_ELEMENTS(*c*) elements, containing the dual solution, is stored.

Discussion

Function LINPROG uses a revised simplex method to solve linear programming problems; i.e., problems of the form

$$\min_{x \in \mathbf{R}^n} c^T x$$

subject to

$$b_l \leq A_x \leq b_u$$

$$x_l \leq x \leq x_u$$

where c is the objective coefficient vector, A is the coefficient matrix, and the vectors b_l, b_u, x_l, and x_u are the lower and upper bounds on the constraints and the variables.

For a complete discussion of the revised simplex method, see Murtagh (1981) or Murty (1983).

This problem can be solved more efficiently.

Example

The linear programming problem in the standard form

$$\min f(x) = -x_0 - 3x_1$$

subject to

$$\begin{aligned} x_0 + x_1 + x_2 &= 1.5 \\ x_0 + x_1 - x_3 &= 0.5 \\ x_0 + x_4 &= 1.0 \\ x_1 + x_5 &= 1.0 \end{aligned}$$

$$x_i \geq 0, \text{ for } i = 0, \ldots, 5$$

is solved.

```
RM, a, 4, 6
```
; Define the coefficients of the constraints.
```
row 0: 1 1 1  0 0 0
row 1: 1 1 0 -1 0 0
row 2: 1 0 0  0 1 0
row 3: 0 1 0  0 0 1
RM, b, 4, 1
```
; Define the right-hand side of the constraints.
```
row 0: 1.5
row 1:  .5
row 2: 1
row 3: 1
RM, c, 6, 1
```
; Define the coefficients of the objective function.
```
row 0: -1
row 1: -3
row 2:  0
row 3:  0
```

```
row 4:   0
row 5:   0

PM, LINPROG(a, b, c), Title = 'Solution'
    ; Call LINPROG and print the solution.
Solution
         0.500000
         1.00000
         0.00000
         1.00000
         0.500000
         0.00000
```

Warning Errors

MATH_PROB_UNBOUNDED — Problem is unbounded.

MATH_TOO_MANY_ITN — Maximum number of iterations exceeded.

MATH_PROB_INFEASIBLE — Problem is infeasible.

Fatal Errors

MATH_NUMERIC_DIFFICULTY — Numerical difficulty occurred. If *float* is currently being used, using *double* may help.

MATH_BOUNDS_INCONSISTENT — Bounds are inconsistent.

QUADPROG Function

Solves a quadratic programming (QP) problem subject to linear equality or inequality constraints.

Usage

result = QUADPROG(*a*, *b*, *g*, *h*)

Input Parameters

a — Two-dimensional matrix containing the linear constraints.

b — One-dimensional matrix of the right-hand sides of the linear constraints.

g — One-dimensional array of the coefficients of the linear term of the objective function.

h — Two-dimensional array of size N_ELEMENTS(*g*) × N_ELEMENTS(*g*) containing the Hessian matrix of the objective function. It must be symmetric positive definite. If *h* is not positive definite, the algorithm attempts to solve the QP problem with *h* replaced by *h* + *diag*I*, such that *h* + *diag*I* is positive definite.

Returned Value

result —The solution *x* of the QP problem.

Input Keywords

Double — If present and nonzero, double precision is used.

Meq — Number of linear equality constraints. If *Meq* is used, then the equality constraints are located at *a* (*i*, *) for *i* = 0, …, *Meq* – 1.

Default: *Meq* = N_ELEMENTS(*a* (*, 0)) *n*; i.e., all constraints are equality constraints

Output Keywords

Diag — Name of the variable into which the scalar, equal to the multiple of the identity matrix added to *h* to give a positive definite matrix, is stored.

Dual — Name of the variable into which an array with N_ELEMENTS(*g*) elements, containing the Lagrange multiplier estimates, is stored.

Obj — Name of the variable into which the optimal object function found is stored.

Discussion

Function QUADPROG is based on M.J.D. Powell's implementation of the Goldfarb and Idnani dual quadratic programming (QP) algorithm for convex QP problems subject to general linear equality/inequality constraints (Goldfarb and Idnani 1983). I.e., problems of the form

$$\min_{x \in \mathbf{R}^n} g^T x + \frac{1}{2} x^T H x$$

subject to

$$A_1 x = b_1$$
$$A_2 x \geq b_2$$

given the vectors b_0, b_1, and g, and the matrices H, A_0, and A_1. Matrix H is required to be positive definite. In this case, a unique x solves the problem, or the constraints are inconsistent. If H is not positive definite, a positive definite perturbation of H is used in place of H. For more details, see Powell (1983, 1985).

If a perturbation of H, $H + \alpha I$, is used in the QP problem, $H + \alpha I$ also should be used in the definition of the Lagrange multipliers.

Example

The QP problem

$$\min f(x) = -x^2_0 + x^2_1 + x^2_2 + x^2_3 + x^2_4 - 2x_1x_2 - 2x_3x_4 - 2x_0$$

subject to

$$x_0 + x_1 + x_2 + x_3 + x_4 = 5$$
$$x_2 - 2x_3 - 2x_4 = -3$$

is solved.

```
RM, a, 2, 5
    ; Define the coefficient matrix A.

row 0: 1 1 1  1  1
row 1: 0 0 1 -2 -2

h = [[2,  0,  0,  0,  0], $
    [0,  2, -2,  0,  0], $
    [0, -2,  2,  0,  0], $
    [0,  0,  0,  2, -2], $
    [0,  0,  0, -2,  2]]
    ; Define the Hessian matrix of the objective function. Notice that since
    ; h is symmetric, the array concatenation operators "[ ]" are used to
    ; define it.

b = [5, -3]
    ; Define b.

g = [ -2, 0, 0, 0, 0]
    ; Define g.

x = QUADPROG(a, b, g, h)
    ; Call QUADPROG.

PM, x
    ; Output solution.

Solution:
      1.00000
      1.00000
      1.00000
      1.00000
      1.00000
```

Warning Errors

MATH_NO_MORE_PROGRESS — Due to the effect of computer rounding error, a change in the variables fails to improve the objective function value. Usually, the solution is close to optimum.

Fatal Errors

MATH_SYSTEM_INCONSISTENT — System of equations is inconsistent. There is no solution.

NONLINPROG Function

Solves a general nonlinear programming problem using the successive quadratic programming (QP) algorithm.

Usage

result = NONLINPROG(*f*, *m*, *n*)

Input Parameters

f — Scalar string specifying a user-supplied procedure to evaluate the function at a given point. Procedure *f* has the following parameters:

> ***m*** — Total number of constraints.
>
> ***meq*** — Number of equality constraints.
>
> ***x*** — One-dimensional array at which point the function is evaluated.
>
> ***active*** — One-dimensional array with *mmax* components indicating the active constraints where *mmax* is the maximum of (1, *m*).
>
> ***f*** — Computed function value at the point *x*. (Output)
>
> ***g*** — One-dimensional array with *mmax* components containing the values of the constraints at point *x*, where *mmax* is the maximum (1, *m*). (Output)

m — Total number of constraints.

n — Number of variables.

Returned Value

result —The solution of the nonlinear programming problem.

Input Keywords

Double — If present and nonzero, double precision is used.

Meq — Number of equality constraints.

> Default: *Meq* = *m*

Ibtype — Scalar indicating the types of bounds on variables.

Ibtype	**Action**
0	User supplies all the bounds.
1	All variables are nonnegative.
2	All variables are nonpositive.
3	User supplies only the bounds on first variable; all other variables have the same bounds.

Default: no bounds are enforced

XGuess — Array with *n* components containing an initial guess of the computed solution.

Default: *XGuess* (*) = 0

Grad — Scalar string specifying a user-supplied procedure to evaluate the gradients at a given point. The procedure specified by *Grad* has the following parameters:

mmax — Maximum of (1, *m*).

m — Total number of constraints.

meg — Number of equality constraints.

x — Array at which point the function is evaluated.

active — Array with *mmax* components indicating the active constraints.

f — Computed function value at the point *x*.

g — Array with *mmax* components containing the values of the constraints at point *x*.

df — Array with *n* components containing the values of the gradient of the objective function. (Output)

dg — Array of size $n \times mmax$ containing the values of the gradients for the active constraints. (Output)

Err_Rel — Final accuracy.

Default: *Err_Rel* = SQRT(ε), where ε is the machine precision

XScale — Array with *n* components containing the reciprocal magnitude of each variable. Keyword *XScale* is used to choose the finite-difference stepsize, *h*.

The *i*-th component of *h* is computed as

$$\sqrt{\varepsilon} * \max(|x_i|, 1/s_i) * \text{sign}(x_i) \quad ,$$

where ε is the machine precision, s = *XScale*, and $\text{sign}(x_i) = 1$ if $x_i \geq 0$; otherwise, $\text{sign}(x_i) = -1$.

Default: *XScale* (*) = 1

Itmax — Maximum number of iterations allowed.

Default: *Itmax* = 200

Output Keywords

Obj — Name of a variable into which a scalar containing the value of the objective function at the computed solution is stored.

Input/Output Keywords

Xlb — Named variable, containing a one-dimensional array with *n* components, containing the lower bounds on the variables. (Input, if *Ibtype* = 0; Output, if *Ibtype* = 1 or 2; Input/Output, if *Ibtype* = 3). If there is no lower bound on a variable, the corresponding *Xlb* value should be set to -10^6.

Default: no lower bounds are enforced on the variables

Xub — Named variable, containing a one-dimensional array with *n* components, containing the upper bounds on the variables. (Input, if *Ibtype* = 0; Output, if *Ibtype* = 1 or 2; Input/Output, if *Ibtype* = 3). If there is no upper bound on a variable, the corresponding *Xub* value should be set to 10^6.

Default: no upper bounds are enforced on the variables

Discussion

Function NONLINPROG, based on the subroutine NLPQL developed by Schittkowski (1986), uses a successive quadratic programming method to solve the general nonlinear programming problem.

The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

subject to

$$g_j(x) = 0, \text{ for } j = 0, \ldots, m_e - 1$$

$$g_j(x) \geq 0, \text{ for } j = m_e, \ldots, m - 1$$

$$(x_l \leq x \leq x_u)$$

where all problem functions are assumed to be continuously differentiable. The method, based on the iterative formulation and solution of quadratic programming (QP) subproblems, obtains these subproblems by using a quadratic approximation of the Lagrangian and by linearizing the constraints. That is,

$$\min_{x \in \mathbf{R}^n} \frac{1}{2} d^T B_k d + \nabla f(x_k)^T d$$

subject to

$$\nabla g_j(x_k)^T d + g_j(x_k) = 0, \text{ for } j = 0, \ldots, m_e - 1$$

$$\nabla g_j(x_k)^T d + g_j(x_k) \geq 0, \text{ for } j = m_e, \ldots, m - 1$$

$$x_l - x_k \leq d \leq x_u - x_k$$

where B_k is a positive definite approximation of the Hessian and x_k is the current iterate. Let d_k be the solution of the subproblem. A line search is used to find a new point x_{k+1}

$$x_{k+1} = x_k + \lambda d_k \qquad \lambda \in (0, 1]$$

such that a "merit function" has a lower function value at the new point. Here, the augmented Lagrange function (Schittkowski 1986) is used as the merit function.

When optimality is not achieved, B_k is updated according to the modified BFGS formula (Powell 1978). Note that this algorithm may generate infeasible points

during the solution process. Therefore, if feasibility must be maintained for intermediate points, this function may not be suitable. For more theoretical and practical details, see Stoer (1985), Schittkowski (1983, 1986), and Gill et al. (1985).

Example

The problem

$$\min F(x) = (x_1 - 2)^2 + (x_2 - 1)^2$$

subject to

$$g_1(x) = x_1 - 2x_2 + 1 = 0$$

$$g_2(x) = -x^2_1/4 - x^2_2 + 1 \geq 0$$

is solved.

```
.RUN
    ; Define the procedure to evaluate the function at a given point.
- PRO f, m, meq, x, active, f, g
- tmp1 = x(0) - 2.
- tmp2 = x(1) - 1.
- f = tmp1^2 + tmp2^2
- g = FLTARR(2)
- IF active(0) THEN g(0) = x(0) - 2. * $
-    x(1) + 1.
- IF active(1) THEN g(1) = -(x(0)^2)/4. - $
-    x(1)^2 + 1.
- END
x = NONLINPROG('f', 2, 2, Meq = 1)
    ; Call NONLINPROG to compute the solution.
PM, x, Title = 'Solution:'
    ; Output the solution.
Solution:
      0.822902
      0.911452
```

Warning Errors

MATH_TOO_MANY_ITN — Maximum number of iterations exceeded.

Fatal Errors

MATH_UPHILL_DIRECTION — Search direction is uphill.

MATH_TOO_MANY_LINESEARCH — Line search took more than five function calls.

MATH_NO_PROGRESS_MADE — Search direction is close to zero.

MATH_QP_INCONSISTENT — Constraints for the QP subproblem are inconsistent.

MINCONGEN Function

Minimizes a general objective function subject to linear equality/inequality constraints.

Usage

result = MINCONGEN(*f*, *a*, *b*, *xlb*, *xub*)

Input Parameters

***f* —** Scalar string specifying a user-supplied function to evaluate the function to be minimized. Function *f* accepts the following input parameters:

> ***x* —** One-dimensional array of length n = N_ELEMENTS(*x*) containing the point at which the function is evaluated.
>
> The return value of this function is the function value at *x*.

***a* —** Two-dimensional array of size ncon by nvar containing the equality constraint gradients in the first *Meq* rows followed by the inequality constraint gradients, where ncon is the number of linear constraints (excluding simple bounds) and nvar is the number of variables. See keyword *Meq* for setting the number of equality constraints.

***b* —** One-dimensional array of size ncon containing the right-hand sides of the linear constraints. Specifically, the constraints on the variables x_i, i = 0, nvar – 1, are $a_{k,0}x_0 + ... + a_{k,nvar-1}x_{nvar-1} = b_k$, k = 0, ..., *Meq* –1 and $a_{k,0}x_0 + ... + a_{k,nvar-1}x_{nvar-1} \le b_k$, k = *Meq*, ..., ncon - 1. Note that the data that define the equality constraints come before the data of the inequalities.

***xlb* —** One-dimensional array of length nvar containing the lower bounds on the variables; choose a very large negative value if a component should be unbounded below or set *xlb*(*i*) = *xub*(*i*) to freeze the *i*-th variable. Specifically, these simple bounds are $xlb(i) \le x_i$, *for* $i = 0, ..., nvar-1$.

***xub* —** One-dimensional array of length nvar containing the upper bounds on the variables; choose a very large positive value if a component should be unbounded above. Specifically, these simple bounds are $x_i \le xub(i)$, for $i = 0, nvar-1$.

Returned Value

***result* —** One-dimensional array of length nvar containing the computed solution.

Input Keywords

***Double* —** If present and nonzero, double precision is used.

***Meq* —** Number of linear equality constraints.

Default: *Meq* = 0

***Xguess* —** One-dimensional array with nvar components containing an initial guess.

Default: *Xguess* = 0

***Grad* —** Scalar string specifying the name of the user-supplied function to compute the gradient at the point *x*. The function accepts the following parameters:

***X* —** One-dimensional array of length nvar.

The return value of this function is a one-dimensional array of length nvar containing the values of the gradient of the objective function.

***Max_Fcn* —** Maximum number of function evaluations.

Default: *Max_Fcn* = 400

***Tolerance* —** The nonnegative tolerance on the first order conditions at the calculated solution.

Default: *Tolerance* = SQRT(ε), where ε is machine epsilon.

Output Keywords

***Num_Active* —** Named variable into which the final number of active constraints is stored.

***Active_Const* —** Named variable into which an one-dimensional array of length *Num_Active* containing the indices of the final active constraints is stored.

***Lagrange_Mult* —** Named variable into which an one-dimensional array of length *Num_Active* containing the Lagrange multiplier estimates of the final active constraints is stored.

***Obj* —** Named variable into which the value of the objective function is stored.

Discussion

The function MINCONGEN is based on M.J.D. Powell's TOLMIN, which solves linearly constrained optimization problems, i.e., problems of the form

$$\min f(x)$$

subject to

$$A_1 x = b_1$$
$$A_2 x \leq b_2$$
$$x_l \leq x \leq x_u$$

given the vectors b_1, b_2, x_l ,and x_u and the matrices A_1 and A_2.

The algorithm starts by checking the equality constraints for inconsistency and redundancy. If the equality constraints are consistent, the method will revise x^0, the initial guess, to satisfy

$$A_1 x = b_1$$

Next, x^0 is adjusted to satisfy the simple bounds and inequality constraints. This is done by solving a sequence of quadratic programming subproblems to minimize the sum of the constraint or bound violations.

Now, for each iteration with a feasible x^k, let J_k be the set of indices of inequality constraints that have small residuals. Here, the simple bounds are treated as inequality constraints. Let I_k be the set of indices of active constraints. The following quadratic programming problem

$$\min f\left(x^k\right) + d^T \nabla f\left(x^k\right) + \frac{1}{2} d^T B^k d$$

subject to

$$a_j d = 0,\ j \in I_k$$

$$a_j d \leq 0,\ j \in J_k$$

is solved to get (d^k, λ^k) where a_j is a row vector representing either a constraint in A_1 or A_2 or a bound constraint on x. In the latter case, the $a_j = e_i$ for the bound constraint $x_i \leq (x_u)_i$ and $a_j = -e_i$ for the constraint $-x_i \leq (x_l)_i$. Here, e_i is a vector with
1 as the i-th component, and zeros elsewhere. Variables λ^k are the Lagrange multipliers, and B^k is a positive definite approximation to the second derivative $\nabla^2 f(x^k)$.

After the search direction d^k is obtained, a line search is performed to locate a better point. The new point $x^{k+1} = x^k + \alpha^k d^k$ has to satisfy the conditions

$$f(x^k + \alpha^k d^k) \leq f(x^k) + 0.1\ \alpha^k\ (d^k)^T\ \nabla\ f(x^k)$$

and

$$(d^K)^T \nabla\ f(x^k + \alpha^k d^k) \leq 0.7\ (d^k)^T \nabla\ f(x^K)$$

The main idea in forming the set J_k is that, if any of the equality constraints restricts the step-length α^k, then its index is not in J_k. Therefore, small steps are likely to be avoided.

Finally, the second derivative approximation B^K, is updated by the BFGS formula, if the condition

$$(d^K)^T \nabla\ f(x^k + \alpha^k d^k) - \nabla\ f(x^k) > 0$$

holds. Let $x^k \leftarrow x^{k+1}$, and start another iteration.

The iteration repeats until the stopping criterion

$$\| \nabla\ f(x^k) - A^k \lambda^K \|_2 \leq \tau$$

is satisfied. Here τ is the supplied tolerance. For more details, see Powell (1988, 1989).

Since a finite difference method is used to approximate the gradient for some single precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high precision arithmetic is recommended. Also, if the gradient can be easily provided, the input keyword *Grad* should be used.

Example 1

In this example, the problem

$$\min f(x) = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 2x_2x_3 - 2x_4x_5 - 2x_1$$
$$\text{subject to } x_1 + x_2 + x_3 + x_4 + x_5 = 5$$
$$x_3 - 2x_4 - 2x_5 = -3$$
$$0 \le x \le 10$$

```
FUNCTION fcn,  x
   f  =  x(0)*x(0) + x(1)*x(1) + x(2)*x(2) + x(3)*x(3) + $
         x(4)*x(4) - 2.0*x(1)*x(2) - 2.0*x(3) * x(4) - $
         2.0*x(0)
   RETURN,  f
END

meq  =  2
a  =  TRANSPOSE([[1.0, 1.0, 1.0, 1.0, 1.0], $
                 [0.0, 0.0, 1.0, -2.0, -2.0]])
b  =  [5.0, -3.0]
xlb  =  FLTARR(5)
xlb(*)  =  0.0
xub  =  FLTARR(5)
xub(*)  =  10.0
   ; Call CMAST_ERR_PRINT to suppress note errors
CMAST_ERR_PRINT, 1
x  =  MINCONGEN('fcn', a, b, xlb, xub, Meq = meq)
PM, x, Title = 'Solution'
Solution
       1.00000
       1.00000
       1.00000
       1.00000
       1.00000
```

Example 2

In this example, the problem from Schittkowski (1987)

$$\min f(x) = -x_0x_1x_2$$

$$\text{subject to } -x_0 - 2x_1 - 2x_2 \le 0$$

$$x_0 + 2x_1 + 2x_2 \le 72$$

$$0 \le x_0 \le 20$$

$$0 \le x_1 \le 11$$

$$0 \le x_2 \le 42$$

is solved with an initial guess of $x_0 = 10$, $x_1 = 10$ and $x_2 = 10$.

```
FUNCTION fcn,  x
   f  =  -x(0)*x(1)*x(2)
   RETURN,  f
END

FUNCTION gradient, x
   g  =  FLTARR(3)
   g(0)  =  -x(1)*x(2)
   g(1)  =  -x(0)*x(2)
   g(2)  =  -x(0)*x(1)
   RETURN, g
END

meq  =  0
a  =  TRANSPOSE([[-1.0, -2.0, -2.0], [1.0, 2.0, 2.0]])
b  =  [0.0, 72.0]
xlb  =  FLTARR(3)
xlb(*)  =  0.0
xub  =  [20.0, 11.
xguess  =  FLTARR(3)
xguess(*)  =  10.0
   Call CMAST_ERR_PRINT to suppress note errors
CMAST_ERR_PRINT, 1
x  =  MINCONGEN('fcn', a, b, xlb, xub, Meq = meq, $
           Grad = 'gradient', Xguess = xguess, Obj = obj)
PM, x, Title = 'Solution'
Solution
```

```
      20.0000
      11.0000
      15.0000
PRINT, "Objective value =", obj
Objective value =     -3300.00
```

9

CHAPTER

Special Functions

Contents of Chapter

Elliptic Integrals

Fresnel Integrals

Airy Functions

Kelvin Functions

Basic Financial Functions

Bond Functions

ERF Function

Evaluates the real error function erf(x). Using a keyword, the inverse error function erf $^{-1}$(x) can be evaluated.

Usage

result = ERF(x)

Input Parameters

x — Expression for which the error function is to be evaluated.

Returned Value

result — The value of the error function erf(x).

Input Keywords

Double — If present and nonzero, double precision is used.

Inverse — Evaluates the real inverse error function erf^{-1}(x). The inverse error function is defined only for $-1 < x < 1$.

Discussion

The error function erf(x) is defined below.

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

All values of x are legal. The inverse error function y = erf $^{-1}$(x) is such that x = erf (y).

Example 1

Plot the error function over [–3, 3].

```
x = 6 * FINDGEN(100)/99 - 3
PLOT, x, ERF(x), XTitle = "x", YTitle = "erf(x)"
```

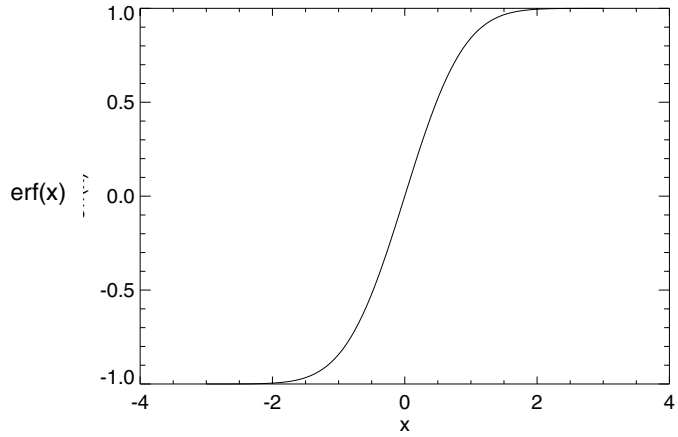


Figure 9-1 Plot of *erf(x)*.

Example 2

Plot the inverse of the error function over (–1, –1).

```
x = 2 * FINDGEN(100)/99 - 1
PLOT, x, ERF(x(1:98), /Inverse), XTitle = "x", $
   YTitle = "erf!E-1!N(x)"
```

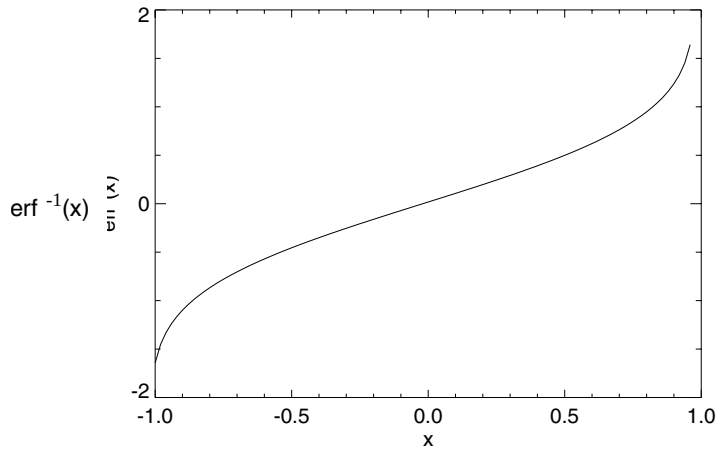


Figure 9-2 Plot of erf $^{-1}(x)$.

ERFC Function

Evaluates the real complementary error function erfc(x). Using a keyword, the inverse complementary error function erfc$^{-1}(x)$ can be evaluated.

Usage

result = ERFC(x)

Input Parameters

x — Expression for which the complementary error function is to be evaluated.

Returned Value

result — The value of the complementary error function erfc(x).

Input Keywords

Double — If present and nonzero, double precision is used.

Inverse — Evaluates the inverse complementary error function erfc$^{-1}(x)$. The parameter must be in the range $0 < x < 2$.

Discussion

The complementary error function erfc(x) is defined as

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$$

where parameter x must not be so large that the result underflows. Approximately, x should be less than

$$[-\ln(\sqrt{\pi}\, s)]^{1/2} ,$$

where s is the smallest representable floating-point number.

The inverse complementary error function $y = \text{erfc}^{-1}(x)$ is such that $x = \text{erfc}(y)$.

Example 1

Plot the complementary error function over [–3, 3].

```
x = FINDGEN(100)/99

PLOT, 6 * x - 3, ERFC(6 * x - 3), XTitle = "x", $
   YTitle = "erfc(x)"
```

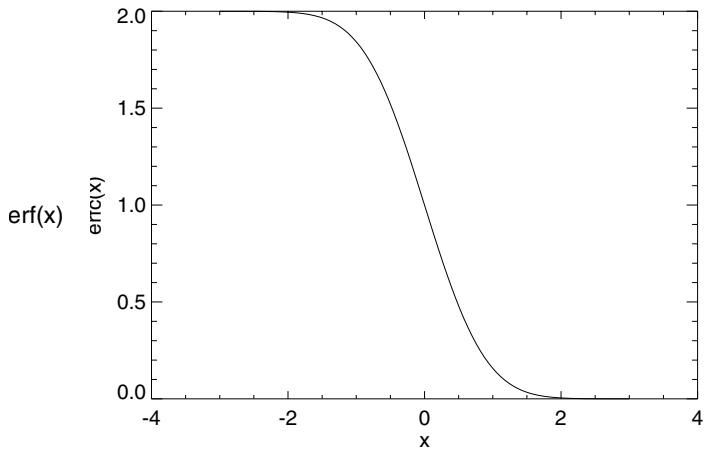


Figure 9-3 Plot of *erf(x)*.

Example 2

Plot the inverse of the complementary error function over (0, 2).

```
x = FINDGEN(100)/99
PLOT, 2 * x(1:98), ERFC(2 * x(1:98), /Inverse), XTitle = "x", $
   YTitle = "erfc!E-1!N(x)"
```

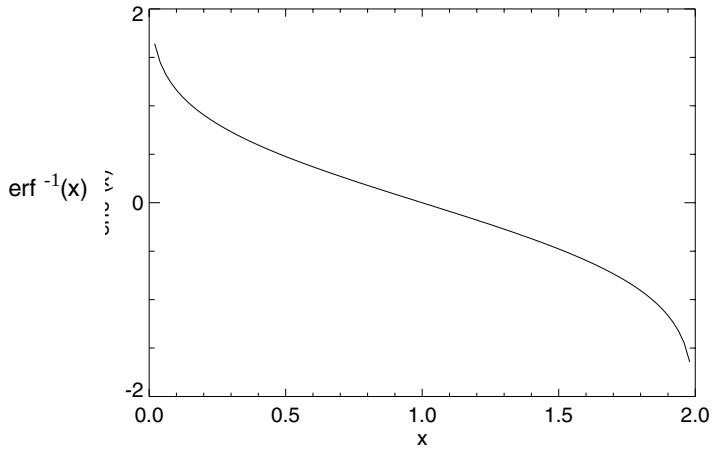


Figure 9-4 Plot of $\text{erfc}^{-1}(x)$.

Alert Errors

MATH_LARGE_ARG_UNDERFLOW — Parameter *x* must not be so large that the result underflows. Very approximately, *x* should be less than

$$2 - \sqrt{\varepsilon/(4\pi)} ,$$

where ε is the machine precision.

Warning Errors

MATH_LARGE_ARG_WARN — Parameter |*x*| should be less than

$$1/(\sqrt{\varepsilon}),$$

where ε is the machine precision, to prevent the answer from being less accurate than half precision.

Fatal Errors

MATH_ERF_ALGORITHM — Algorithm failed to converge.

MATH_SMALL_ARG_OVERFLOW — Computation of

$$e^{x^2}\,\mathrm{erfc}(x)$$

must not overflow.

MATH_REAL_OUT_OF_RANGE — Function is defined only for 0 < x < 2.

BETA Function

Evaluates the real beta function $\beta(x, y)$.

Usage

result = BETA(*x*, *y*)

Input Parameters

x — First beta parameter. It must be positive.

y — Second beta parameter. It must be positive.

Returned Value

result — The value of the beta function $\beta(x, y)$. If no result can be computed, then NaN (Not a Number) is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

The beta function, $\beta(x, y)$, is defined as

$$\beta(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)} = \int_0^1 t^{x-1}(1-t)^{y-1}dt$$

requiring that $x > 0$ and $y > 0$. It underflows for large parameters.

Example

Plot the beta function over $[\varepsilon, 1/4 + \varepsilon]$ x $[\varepsilon, 1/4 + \varepsilon]$ for $\varepsilon = 0.01$.

```
x = 1e-2 + .25 * FINDGEN(25)/24
y = x
b = FLTARR(25, 25)
FOR i = 0, 24 DO b(i, *) = BETA(x(i), y)
    ; Compute values of the beta function.
SURFACE, b, x, y, XTitle = 'X', YTitle = 'Y', Az = 320, ZAxis =
   2
        ; Plot the computed values as a surface and rotate the plot
        ; using the keyword Az.
```

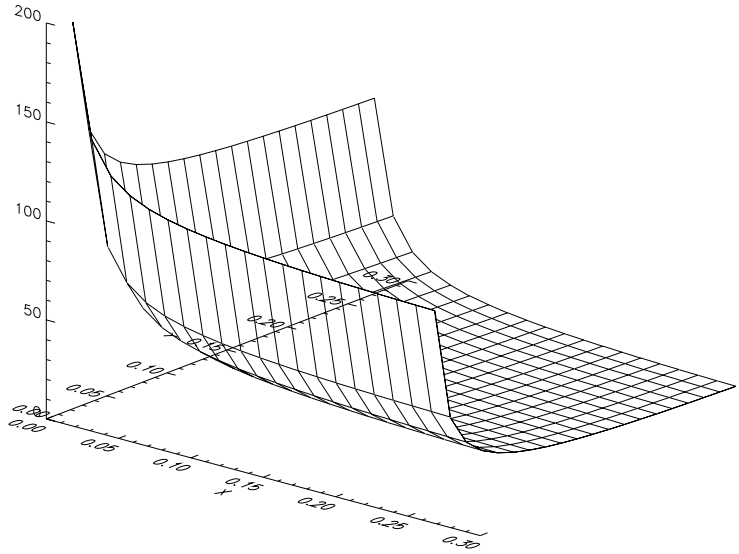


Figure 9-5 Plot of the real beta function.

Alert Errors

MATH_BETA_UNDERFLOW — Parameters must not be so large that the result underflows.

Fatal Errors

MATH_ZERO_ARG_OVERFLOW — One of the parameters is so close to zero that the result overflows.

LNBETA Function

Evaluates the logarithm of the real beta function ln $\beta(x, y)$.

Usage

result = LNBETA(*x*, *y*)

Input Parameters

x — First argument of the beta function. It must be positive.

y — Second argument of the beta function. It must be positive.

Returned Value

result — The value of the logarithm of the beta function $\beta(x, y)$.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

The beta function, $\beta(x, y)$, is defined as

$$\beta(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x + y)} = \int_0^1 t^{x-1}(1 - t)^{y-1} dt$$

and LNBETA returns ln $\beta(x, y)$. The logarithm of the beta function requires that $x > 0$ and $y > 0$. It can overflow for very large parameters.

Example

Evaluate the log of the beta function ln β (0.5, 0.2).

```
PM, LNBETA(.5, .2)
       1.83556
```

Warning Errors

`MATH_X_IS_TOO_CLOSE_TO_NEG_1` — Result is accurate to less than one precision because the expression $-x / (x + y)$ is too close to –1.

BETAI Function

Evaluates the real incomplete beta function.

Usage

result = BETAI(*x, a, b*)

Input Parameters

x — Upper limit of integration.

a — First beta distribution parameter.

b — Second beta distribution parameter.

Returned Value

result — The value of the incomplete beta function.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

The incomplete beta function is defined as

$$I_x(a, b) = \frac{\beta_x(a, b)}{\beta(a, b)} = \frac{1}{\beta(a, b)} \int_0^x t^{a-1}(1-t)^{b-1} dt$$

requiring that $0 \le x \le 1$, $a > 0$, and $b > 0$. It underflows for sufficiently small x and large a. This underflow is not reported as an error. Instead, the value zero is returned.

Example

In this example, $I_{0.61}(2.2, 3.7)$ is computed and printed.

```
PM, BETAI(.61, 2.2, 3.7)
      0.882172
```

GAMMA Function

Evaluates the real gamma function $\Gamma(x)$.

Usage

result = GAMMA(*x*)

Input Parameters

x — Expression for which the gamma function is to be evaluated.

Returned Value

result — The value of the gamma function $\Gamma(x)$.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

The gamma function, $\Gamma(x)$, is defined as follows:

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$

For $x < 0$, the above definition is extended by analytic continuation.

The gamma function is not defined for integers less than or equal to zero. It underflows for

$$x << 0$$

and overflows for large x. It also overflows for values near negative integers.

Example

In this example, $\Gamma(1.5)$ is computed and printed.

```
PM, GAMMA(1.5)
      0.886227
```

Alert Errors

MATH_SMALL_ARG_UNDERFLOW — Parameter x must be large enough that $\Gamma(x)$ does not underflow. The underflow limit occurs first for parameters close to large negative half integers. Even though other parameters away from these half integers may yield machine-representable values of $\Gamma(x)$, such parameters are considered illegal. Users who need such values should use the log $\Gamma(x)$ function LNGAMMA.

Warning Errors

MATH_NEAR_NEG_INT_WARN — Result is accurate to less than one-half precision because x is too close to a negative integer.

Fatal Errors

MATH_ZERO_ARG_OVERFLOW — Parameter for the gamma function is too close to zero.

MATH_NEAR_NEG_INT_FATAL — Parameter for the function is too close to a negative integer.

MATH_LARGE_ARG_OVERFLOW — Function overflows because x is too large.

MATH_CANNOT_FIND_XMIN — Algorithm used to find x_{min} failed. This error should never occur.

MATH_CANNOT_FIND_XMAX — Algorithm used to find x_{max} failed. This error should never occur.

LNGAMMA Function

Evaluates the logarithm of the absolute value of the gamma function $\log|\Gamma(x)|$.

Usage

result = LNGAMMA(*x*)

Input Parameters

x — Expression for which the logarithm of the absolute value of the gamma function is to be evaluated.

Returned Value

result — The value of the logarithm of gamma function $\log|\Gamma(x)|$.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

The logarithm of the absolute value of the gamma function $\log|\Gamma(x)|$ is computed.

Example

In this example, $\log|\Gamma(3.5)|$ is computed and printed.

```
PM, LNGAMMA(3.5)
       1.20097
```

Warning Errors

MATH_NEAR_NEG_INT_WARN — Result is accurate to less than one-half precision because x is too close to a negative integer.

Fatal Errors

MATH_NEGATIVE_INTEGER — Parameter for the function cannot be a negative integer.

MATH_NEAR_NEG_INT_FATAL — Parameter for the function is too close to a negative integer.

MATH_LARGE_ABS_ARG_OVERFLOW — Parameter $|x|$ must not be so large that the result overflows.

GAMMAI Function

Evaluates the incomplete gamma function $\gamma(a, x)$.

Usage

result = GAMMAI(*a*, *x*)

Input Parameters

a — Integrand exponent parameter. It must be positive.

x — Upper limit of integration. It must be nonnegative.

Returned Value

result — The value of the incomplete gamma function $\gamma(a, x)$.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

The incomplete gamma function, $\gamma(a, x)$, is defined as follows:

$$\gamma(a, x) = \int_0^x t^{a-1} e^{-t} dt$$

The incomplete gamma function is defined only for $a > 0$. Although $\gamma(a, x)$ is well-defined for $x > -infinity$, this algorithm does not calculate $\gamma(a, x)$ for negative x. For large a and sufficiently large x, $\gamma(a, x)$ may overflow. Gamma function $\gamma(a, x)$ is bounded by $\Gamma(a)$, and users may find this bound a useful guide in determining legal values for a.

Example

Plot the incomplete gamma function over [0.1, 1.1] **x** [0, 4].

```
x = 4. * FINDGEN(25)/24
a = 1e-1 + FINDGEN(25)/24
b = FLTARR(25, 25)
FOR i = 0, 24 DO b(i, *) = GAMMAI(a(i), x)
!P.Charsize = 2.5
SURFACE, b, a, x, XTitle = 'a', YTitle = 'X'
```

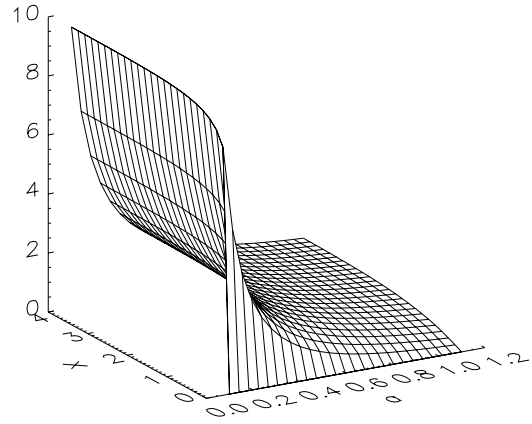


Figure 9-6 Plot of incomplete gamma function.

Fatal Errors

MATH_NO_CONV_200_TS_TERMS — Function did not converge in 200 terms of Taylor series.

MATH_NO_CONV_200_CF_TERMS — Function did not converge in 200 terms of the continued fraction.

BESSI Function

Evaluates a modified Bessel function of the first kind with real order and real or complex parameters.

Usage

result = BESSI(*order, z*)

Input Parameters

order — Real parameter specifying the desired order. Parameter *order* must be greater than –1 / 2.

z — Real or complex parameter for which the Bessel function is to be evaluated.

Returned Value

result — The desired value of the modified Bessel function.

Input Keywords

Double — If present and nonzero, double precision is used.

Sequence — If present and nonzero, a one-dimensional array of length n containing the values of the Bessel function through the series is returned by BESSI, where n = N_ELEMENTS(*Sequence*). The *i*-th element of this array is the Bessel function of order (*order* + *i*) at *z* for *i* = 0, ... (n – 1).

Discussion

Function BESSI evaluates a modified Bessel function of the first kind with real order and real or complex parameters. The data type of the returned value is always complex.

The Bessel function, $I_v(z)$, is defined as follows:

$$I_v(z) = e^{-v\pi i/2} J_v(ze^{\pi i/2}) \quad \text{for} \quad -\pi < \arg z \leq \frac{\pi}{2}$$

For large parameters, *z*, Temme's (1975) algorithm is used to find $I_v(z)$. The $I_v(z)$ values are recurred upward (if this is stable). This involves evaluating a continued fraction. If this evaluation fails to converge, the answer may not be accurate. For moderate and small parameters, Miller's method is used.

Example

In this example, $J_{0.3 + v-1}(1.2 + 0.5i)$, $v = 1, \ldots 4$ is computed and printed first by calling BESSI four times in a row, then by using the keyword *Sequence*.

```
z = COMPLEX(1.2, .5)
FOR i = 0, 3 DO PM, BESSI(i + .3, z)
(    1.16339,  0.396301)
(   0.447264,  0.332142)
(  0.0821799,  0.127165)
( 0.00577678, 0.0286277)
PM, BESSI(.3, z, Sequence = 4), $
   Title = 'With SEQUENCE:'
With SEQUENCE:
(    1.16339,  0.396301)
(   0.447264,  0.332142)
(  0.0821799,  0.127165)
( 0.00577678, 0.0286277)
```

BESSJ Function

Evaluates a Bessel function of the first kind with real order and real or complex parameters.

Usage

result = BESSJ(*order, z*)

Input Parameters

order — Real parameter specifying the desired order. Parameter *order* must be greater than –1 / 2.

z — Real or complex parameter for which the Bessel function is to be evaluated.

Returned Value

result — The desired value of the Bessel function.

Input Keywords

Double — If present and nonzero, double precision is used.

Sequence — If present and nonzero, a one-dimensional array of length n containing the values of the Bessel function through the series is returned by BESSJ, where n = NELEMENTS(*Sequence*). The *i*-th element of this array is the Bessel function of order (*order* + *i*) at *z* for *i* = 0, ... (n – 1).

Discussion

Function BESSJ evaluates a Bessel function of the first kind with real order and real or complex parameters. The data type of the returned value is always complex.

The Bessel function, $J_\nu(z)$, is defined as follows:

$$J_\nu(z) = \frac{1}{\pi}\int_0^\pi \cos(z\sin\theta - \nu\theta)d\theta - \frac{\sin(\gamma\pi)}{\pi}\int_0^\infty e^{z\sinh t - \nu t}dt$$

for

$$|\arg z| < \frac{\pi}{2}$$

This function is based on the code BESSCC of Barnett (1981) and Thompson and Barnett (1987). This code computes $J_v(z)$ from the modified Bessel function $I_v(z)$, using the following relation with

$$\rho = e^{i\pi/2}:$$

$$Y_v(z) = \begin{cases} \rho I_v(z/\rho) & \text{for} \quad -\pi/2 < \arg z \le \pi \\ \rho^3 I_v(\rho^3 z) & \text{for} \quad -\pi < \arg z \le \pi/2 \end{cases}$$

Example

In this example, $J_{0.3+v-1}(1.2 + 0.5i)$, $v = 1, ..., 4$ is computed and printed.

```
z = COMPLEX(1.2, .5)
FOR i = 0, 3 DO PM, BESSJ(i + .3, z)
(   0.773756, -0.106925)
(   0.400001,  0.158598)
(  0.0867063, 0.0920276)
( 0.00844932, 0.0239868)
PM, BESSJ(.3, z, Sequence = 4), $
   Title = 'With SEQUENCE:'
With SEQUENCE:
(   0.773756, -0.106925)
(   0.400001,  0.158598)
(  0.0867063, 0.0920276)
( 0.00844932, 0.0239868)
```

BESSK Function

Evaluates a modified Bessel function of the second kind with real order and real or complex parameters.

Usage

result = BESSK(*order, z*)

Input Parameters

order — Real parameter specifying the desired order. Parameter *order* must be greater than –1 /2.

z — Real or complex parameter for which the Bessel function is to be evaluated.

Returned Value

result — The desired value of the modified Bessel function.

Input Keywords

Double — If present and nonzero, double precision is used.

Sequence — If present and nonzero, a one-dimensional array of length n containing the values of the Bessel function through the series is returned by BESSK, where n = NELEMENTS(*Sequence*). The *i*-th element of this array is the Bessel function of order (*order* + *i*) at *z* for *i* = 0, ... (n – 1).

Discussion

Function BESSK evaluates a modified Bessel function of the second kind with real order and real or complex parameters. The data type of the returned value is always complex.

The Bessel function, $K_\nu(z)$, is defined as follows:

$$K_\nu(z) = \frac{\pi}{2} e^{\nu\pi i/2}[iJ_\nu(iz) - Y_\nu(iz)] \qquad \text{for } -\pi < \arg z \le \frac{\pi}{2}$$

This function is based on the code BESSCC of Thompson (1981) and Thompson and Barnett (1987). For moderate or large parameters, *z*, Temme's (1975)

algorithm is used to find $K_v(z)$. This involves evaluating a continued fraction. If this evaluation fails to converge, the answer may not be accurate. For small z, a Neumann series is used to compute $K_v(z)$. Upward recurrence of the $K_v(z)$ is always stable.

Example

In this example, $K_{0.3+v-1}(1.2 + 0.5i)$, $v = 1, ..., 4$ is computed and printed.

```
z = COMPLEX(1.2, .5)
FOR i = 0, 3 DO PM, BESSK(i + .3, z)
( 0.245546, -0.199599)
( 0.335637, -0.362005)
( 0.586718,  -1.12610)
( 0.719457,  -4.83864)
PM, BESSK(.3, z, Sequence = 4), $
   Title = 'With SEQUENCE:'
With SEQUENCE:
( 0.245546, -0.199599)
( 0.335637, -0.362005)
( 0.586718,  -1.12610)
( 0.719456,  -4.83864)
```

BESSY Function

Evaluates a Bessel function of the second kind with real order and real or complex parameters.

Usage

result = BESSY(*order*, *z*)

Input Parameters

order — Real parameter specifying the desired order. Parameter *order* must be greater than –1 / 2.

z — Real or complex parameter for which the Bessel function is to be evaluated.

Returned Value

result — The desired value of the modified Bessel function.

Input Keywords

Double — If present and nonzero, double precision is used.

Sequence — If present and nonzero, a one-dimensional array of length n containing the values of the Bessel function through the series is returned by BESSY, where n = NELEMENTS(*Sequence*). The *i*-th element of this array is the Bessel function of order (*order* + *i*) at *z* for *i* = 0, ... (n – 1).

Discussion

Function BESSY evaluates a Bessel function of the second kind with real order and real or complex parameters. The data type of the returned value is always complex.

The Bessel function, $Y_v(z)$, is defined as follows:

$$Y_\nu(z) = \frac{1}{\pi}\int_0^\pi \sin(z\sin\theta - \nu\theta)d\theta - \frac{\sin(\gamma\pi)}{\pi}\int_0^\infty [e^{\nu t} + e^{-\nu t}\cos(\nu t)]e^{z\sinh t}dt$$

for $|\arg z| < \frac{\pi}{2}$

This function is based on the code BESSCC of Thompson (1981) and Thompson and Barnett (1987). This code computes $Y_v(z)$ from the modified Bessel functions $I_v(z)$ and $K_v(z)$, using the following relation:

$$Y_\nu(z) = e^{(\nu+1)\pi i/2}I_\nu(z) - \frac{2}{\pi}e^{-\nu\pi i/2}K_\nu(z) \quad \text{for} -\pi < \arg z \le \frac{\pi}{2}$$

Example

In this example, $Y_{0.3+v-1}(1.2 + 0.5i)$, $v = 1, ..., 4$ is computed and printed.

```
z = COMPLEX(1.2, .5)
FOR i = 0, 3 DO PM, BESSY(i + .3, z)
( -0.0131453, 0.379593)
(  -0.715533, 0.338082)
(   -1.04777, 0.794969)
(   -1.62487,  3.68447)
PM, BESSY(.3, z, Sequence = 4), Title = 'With SEQUENCE:'
With SEQUENCE:
( -0.0131453, 0.379593)
(  -0.715533, 0.338082)
(   -1.04777, 0.794969)
(   -1.62487,  3.68447)
```

BESSI_EXP Function

Evaluates the exponentially scaled modified Bessel function of the first kind of orders zero and one.

Usage

result = BESSI_EXP(*order, x*)

Input Parameters

order — Order of the function. The order must be either zero or one.

x — Argument for which the function value is desired.

Returned Value

result — The value of the exponentially scaled modified Bessel function of the first kind of order zero or one evaluated at *x*.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

If *order* is zero, the Bessel function is $I_0(x)$ is defined to be

$$I_0(x) = \frac{1}{\pi}\int_0^{\pi} cos(x\,cos\theta)\,d\theta$$

If *order* is one, the function $I_1(x)$ is defined to be

$$I_1(x) = \frac{1}{\pi}\int_0^{\pi} e^{x\,cos\theta}\,cos\theta\,d\theta$$

If *order* is one then BESSI_EXP underfows if $|x| / 2$ underflows.

Example

The expression $e^{-4.5}I_0$ (4.5) is computed directly by calling BESSI_EXP and indirectly by calling BESSI. The absolute difference is printed. For large *x*, the internal scaling provided by BESSI_EXP avoids overflow that may occur in BESSI.

Output

```
ans = BESSI_EXP(0, 4.5)
error = ABS(ans - EXP(-4.5)*BESSI(0, 4.5))
PRINT, ans
      0.194198
PRINT, "Error =", error
Error =    4.4703484e-08
```

BESSK_EXP Function

Evaluates the exponentially scaled modified Bessel function of the third kind of orders zero and one.

Usage

result = BESSK_EXP(*order, x*)

Input Parameters

order — Order of the function. The order must be either zero or one.

x — Argument for which the function value is desired.

Returned Value

result — The value of the exponentially scaled Bessel function $e^x K_0(x)$ or $e^x K_1(x)$

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

If *order* is zero, the Bessel function $K_0(x)$ is defined to be

$$K_0(x) = \int_0^\infty \cos(x \sin t)\, dt$$

If *order* is one, the value of the Bessel function $K_1(x)$

$$I_1(x) = \frac{1}{\pi} \int_0^\pi e^{x \cos\theta} \cos\theta \, d\theta$$

The argument *x* must be greater than zero for the result to be defined.

Example

The expression

$$\sqrt{e} K_0(0.5)$$

is computed directly by calling BESSK_EXP and indirectly by calling BESSK. The absolute difference is printed. For large *x*, the internal scaling provided by BESSK_EXP avoids underflow that may occur in BESSK.

```
ans = BESSK_EXP(0, 0.5)
error = ABS(ans - (EXP(0.5))*BESSK(0, 0.5))
PRINT, ans
       1.52411
PRINT, "Error =", error
Error =    1.1920929e-07
```

Fatal Errors

MATH_SMALL_ARG_OVERFLOW — The argument *x* must be large enough ($x > \max(1/b, s)$ where *s* is the smallest representable positive number and *b* is the largest repesentable number) that $K_1(x)$ does not overflow.

ELK Function

Evaluates the complete elliptic integral of the kind $K(x)$.

Usage

result = ELK(*x*)

Input Parameters

x — Argument for which the function value is desired.

Returned Value

result — The complete elliptic integral $K(x)$.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

The complete elliptic integral of the first kind is defined to be

$$K(x) = \int_0^{\pi/2} \frac{d\theta}{\left[1 - x \sin^2\theta\right]^{1/2}} \text{ for } 0 \le x < 1$$

The argument x must satisfy $0 \le x < 1$; otherwise, ELK returns the largest representable floating-point number.

The function $K(x)$ is computed using the routine ELRF (page 383) and the relation $K(x) = R_F(0, 1 - x, 1)$.

Example

The integral $K(0)$ is evaluated.

```
PRINT,  ELK(0.0)
        1.57080
```

ELE Function

Evaluates the complete elliptic integral of the second kind $E(x)$.

Usage

result = ELE(*x*)

Input Parameters

x — Argument for which the function value is desired.

Returned Value

result — The complete elliptic integral $E(x)$.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

The complete elliptic integral of the second kind is defined to be

$$E(x) = \int_0^{\pi/2} \left[1 - x \sin^2\theta\right]^{1/2} d\theta \text{ for } 0 \le x < 1$$

The argument x must satisfy $0 \le x < 1$; otherwise, ELE returns the largest representable floating-point number.

The function $E(x)$ is computed using the routine ELRF (page 383) and ELRD (page 384). The computation is done using the relation

$$E(x) = R_F(0, 1-x, 1) - \frac{x}{3} R_D(0, 1-x, 1)$$

Example

The integral $E(0.33)$ is evaluated.

```
PRINT, ELE(0.33)
```

```
1.43183
```

ELRF Function

Evaluates Carlson's elliptic integral of the first kind $R_F(x, y, z)$.

Usage

result = ELRF(*x*, *y*, *z*)

Input Parameters

x — First argument for which the function value is desired. It must be nonnegative.

y — Second argument for which the function value is desired. It must be nonnegative.

z — Third argument for which the function value is desired. It must be nonnegative.

Returned Value

result — The complete elliptic integral $R_F(x, y, z)$

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Carlson's elliptic integral of the second kind is defined to be

$$R_F(x,y,z) = \frac{1}{2}\int_0^\infty \frac{dt}{\left[(t+x)(t+y)(t+z)\right]^{1/2}}$$

The arguments must be nonnegative and less than or equal to $b/5$. In addition, $x + y$, $x + z$, and $y + z$ must be greater than or equal to $5s$. Should any of these conditions fail, ELRF is set to b. Here, b is the largest and is the smallest representable number.

The function ELRF is based on the code by Carlson and Notis (1981) and the work of Carlson (1979).

Example

The integral $R_F(0, 1, 2)$ is computed.

```
PRINT, ELRF(0.0, 1.0, 2.0)
       1.31103
```

ELRD Function

Evaluates Carlson's elliptic integral of the second kind $R_D(x, y, z)$.

Usage

result = ELRD(*x*, *y*, *z*)

Input Parameters

x — First argument for which the function value is desired. It must be nonnegative.

y — Second argument for which the function value is desired. It must be nonnegative.

z — Third argument for which the function value is desired. It must be positive.

Returned Value

result — The complete elliptic integral $R_D(x, y, z)$

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Carlson's elliptic integral of the second kind is defined to be

$$R_D(x,y,z) = \frac{3}{2}\int_0^\infty \frac{dt}{\left[(t+x)(t+y)(t+z)^3\right]^{1/2}}$$

The arguments must be nonnegative and less than or equal to $0.69(-ln\varepsilon)^{1/9}s^{-2/3}$ where e is the machine precision, *s* is the smallest representable positive number. Furthermore, $x + y$ and z must be greater than $\max\{3s^{2/3}, 3/b^{2/3}\}$, where *b* is the largest floating point number. If any of these conditions is false, then ELRD returns *b*.

The function ELRD is based on the code by Carlson and Notis (1981) and the work of Carlson (1979).

Example

The integral $R_D(0, 2, 1)$ is computed.

```
PRINT, ELRD(0.0, 2.0, 1.0)
       1.79721
```

ELRJ Function

Evaluates Carlson's elliptic integral of the third kind $R_J(x, y, z, \rho)$.

Usage

result = ELRJ(*x*, *y*, *z*, *rho*)

Input Parameters

x — First argument for which the function value is desired. It must be nonnegative.

y — Second argument for which the function value is desired. It must be nonnegative.

z — Third argument for which the function value is desired. It must be positive.

rho — Fourth argument for which the function value is desired. It must be positive.

Returned Value

result — The complete elliptic integral $R_J(x, y, z, \rho)$.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Carlson's elliptic integral of the third kind is defined to be

$$R_J(x,y,z,\rho) = \frac{3}{2}\int_0^\infty \frac{dt}{\left[(t+x)(t+y)(t+z)(t+\rho)^2\right]^{1/2}}$$

The arguments must be nonnegative. In addition, $x + y$, $x + z$, $y + z$ and ρ must be greater than or equal to $(5s)^{1/3}$ and less than or equal to $0.3(b/5)^{1/3}$, where s is the smallest representable floating-point number. Should any of these conditions fail ELRJ is set to b, the largest floating-point number.

The function ELRJ is based on the code by Carlson and Notis (1981) and the work of Carlson (1979).

Example

The integral R_J (2, 3, 4, 5) is computed.

```
PRINT, ELRJ(2.0, 3.0, 4.0, 5.0)
      0.142976
```

ELRC Function

Evaluates an elementary integral from which inverse circular functions, logarithms and inverse hyperbolic functions can be computed.

Usage

result = ELRC(*x*, *y*)

Input Parameters

x — First argument for which the function value is desired. It must be nonnegative and must satisfy the conditions given below.

y — Second argument for which the function value is desired. It must be nonnegative and must satisfy the conditions given below.

Returned Value

result — The elliptic integral R_C (*x*, *y*).

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Carlson's elliptic integral of the third kind is defined to be

$$R_C(x,y) = \frac{1}{2}\int_0^\infty \frac{dt}{\left[(t+x)(t+y)^2\right]^{1/2}}$$

The argument x must be nonnegative, y must be positive, and $x + y$ must be less than or equal to $b/5$ and greater than or equal to $5s$. If any of these conditions are false, the ELRC is set to b. Here, b is the largest and s is the smallest representable floating-point number.

The function ELRC is based on the code by Carlson and Notis (1981) and the work of Carlson (1979).

Example

The integral R_C (2.25, 2) is computed.

```
PRINT, ELRC(2.25, 2.0)
      0.693147
```

FRESNEL_COSINE Function

Evaluates the cosine Fresnel integral.

Usage

result = FRESNEL_COSINE(*x*)

Input Parameters

x — Argument for which the function value is desired.

Returned Value

result — The value of the cosine Fresnel integral evaluated at *x*.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

The cosine Fresnel integral is defined to be

$$C(x) = \int_0^x \cos\left(\frac{\pi}{2} t^2\right) dt$$

Example

The Fresnel integral $C(1.75)$ is evaluated.

```
PRINT, FRESNEL_COSINE(1.75)
      0.321935
```

FRESNEL_SINE Function

Evaluates the sine Fresnel integral.

Usage

result = FRESNEL_SINE(*x*)

Input Parameters

x — Argument for which the function value is desired.

Returned Value

result — The value of the sine Fresnel integral evaluated at *x*.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

The sine Fresnel integral is defined to be

$$S(x) = \int_0^x \sin\left(\frac{\pi}{2} t^2\right) dt$$

Example

The Fresnel integral $S(1.75)$ is evaluated.

```
PRINT, FRESNEL_SINE(1.75)
      0.499385
```

AIRY_AI Function

Evaluates the Airy function.

Usage

result = AIRY_AI(*x*)

Input Parameters

x — Argument for which the function value is desired.

Returned Value

result — The value of the Airy function evaluated at *x*, Ai(*x*).

Input Keywords

Double — If present and nonzero, double precision is used.

Derivative — If present and nonzero, then the derivative of the Airy function is computed.

Discussion

The airy function Ai(x) is defined to be

$$Ai(x) = \frac{1}{\pi}\int_0^{\infty} cos(xt + \frac{1}{3}t^3)dt = \sqrt{\frac{x}{3\pi^2}}K_{1/3}(\frac{2}{3}x^{3/2})$$

The Bessel function $K_v(x)$ is defined on page 374.

If $x < -1.31\varepsilon^{-2/3}$, then the answer will have no precision. If $x < -1.31\varepsilon^{-1/3}$, the answer will be less accurate than half precision. Here ε is the machine precision.

x should be less than x_{max} so the answer does not underflow. Very approximately, $x_{max} = \{-1.5\ln s\}^{2/3}$, where s = the smallest representable positive number.

If the keyword *Derivative* is set, then the airy function Ai′(x) is defined to be the derivative of the Airy function, Ai(x) (see page 390). If $x < -1.31\varepsilon^{-2/3}$, then the answer will have no precision. If $x < -1.31\varepsilon^{-1/3}$, the answer will be less accurate than half precision. Here ε is the machine precision. x should be less than x_{max} so that the answer does not underflow. Very approximately, $x_{max} = \{-1.51\ln s\}$, where s is the smallest representable positive number.

Example

In this example, Ai(–4.9) and Ai′(–4.9) are evaluated.

```
PRINT, AIRY_AI(-4.9)
      0.374536
PRINT, AIRY_AI(-4.9, /Derivative)
      0.146958
```

AIRY_BI Function

Evaluates the Airy function of the second kind.

Usage

result = AIRY_BI(*x*)

Input Parameters

x — Argument for which the function value is desired.

Returned Value

result — The value of the Airy function evaluated at *x*, Bi(*x*).

Input Keywords

Double — If present and nonzero, double precision is used.

Derivative — If present and nonzero, then the derivative of the Airy function of the second kind is computed.

Discussion

The airy function Bi(*x*) is defined to be

$$Bi(x) = \frac{1}{\pi}\int_0^\infty exp(xt - \frac{1}{3}t^3)dt + \frac{1}{\pi}\int_0^\infty sin(xt + \frac{1}{3}t^3)dt$$

It can also be expressed in terms of modified Bessel functions of the first kind, $I_v(x)$, and Bessel functions of the first kind $J_v(x)$ (see BESSI (page 370), and BESSJ (page 372)):

$$Bi(x) = \sqrt{\frac{x}{3}}\left[I_{-1/3}(\frac{2}{3}x^{3/2}) + I_{1/3}(\frac{2}{3}x^{3/2}) \right] for\ x > 0$$

and

$$Bi(x) = \sqrt{\frac{-x}{3}}\left[J_{-1/3}(\frac{2}{3}|x|^{3/2}) - J_{1/3}(\frac{2}{3}|x|^{3/2}) \right] \text{ for } x < 0$$

Here ε is the machine precision. If $x < -1.31\varepsilon^{-2/3}$, then the answer will have no precision. If $x < -1.31\varepsilon^{-1/3}$, the answer will be less accurate than half precision. In addition, x should not be so large that $exp[(2/3)x^{3/2}]$ overflows.

If the keyword *Derivative* is set, the airy function Bi′(x) is defined to be the derivative of the Airy function of the second kind, Bi(x) (see page 392). If $x < -1.31\varepsilon^{-2/3}$, then the answer will have no precision. If $x < -1.31\varepsilon^{-1/3}$, the answer will be less accurate than half precision. Here ε is the machine precision. In addition, x should not be so large that $exp[(2/3)x^{3/2}]$ overflows.

Example

In this example, Bi(–4.9) and Bi′(-4.9) are evaluated.

```
PRINT, AIRY_BI(-4.9)
     -0.0577468
PRINT, AIRY_BI(-4.9, /Derivative)
      0.827219
```

KELVIN_BER0 Function

Evaluates the Kelvin function of the first kind, ber, of order zero.

Usage

result = KELVIN_BER0(*x*)

Input Parameters

x — Argument for which the function value is desired.

Returned Value

result — The value of the Kelvin function of the first kind, ber, of order zero evaluated at *x*.

Input Keywords

Double — If present and nonzero, double precision is used.

Derivative — If present and nonzero, then the derivative of the Kelvin function of the first kind, ber, of order zero evaluated at *x* is computed.

Discussion

The Kelvin function $\text{ber}_0(x)$ is defined to be $\Re J_0(xe^{3\pi\iota/4})$. The Bessel function $J_0(x)$ is defined

$$J_0(x) = \frac{1}{\pi}\int_0^{\pi} cos(x\,sin\,\theta)d\theta$$

If the keyword *Derivative* is set, the function $\text{ber}_0'(x)$ is defined to be

$$\frac{d}{dx}\text{ber}_0(x)$$

If $|x| > 119$, NaN is returned.

The function KELVIN_BER0 is based on the work of Burgoyne (1963).

Example

In this example, $ber_0(0.4)$ and $ber_0'(0.6)$ are evaluated.

```
PRINT, KELVIN_BER0(0.4)
      0.999600
PRINT, KELVIN_BER0(0.6, /DERIVATIVE)
    -0.0134985
```

KELVIN_BEI0 Function

Evaluates the Kelvin function of the first kind, bei, of order zero.

Usage

result = KELVIN_BEI0(*x*)

Input Parameters

x — Argument for which the function value is desired.

Returned Value

result — The value of the Kelvin function of the first kind, bei, of order zero evaluated at *x*.

Input Keywords

Double — If present and nonzero, double precision is used.

Derivative — If present and nonzero, then the derivative of the Kelvin function of the first kind, bei, of order zero evaluated at *x* is computed.

Discussion

The Kelvin function $\text{bie}_0(x)$ is defined to be $\Im J_0(xe^{3\pi i/4})$. The Bessel function $J_0(x)$ is defined

$$J_0(x) = \frac{1}{\pi}\int_0^{\pi} \cos(x\sin\theta)d\theta$$

In KELVIN_BEI0, x must be less than 119.

If the keyword *Derivative* is set, the function $\text{bei}_0'(x)$ is defined to be

$$\frac{d}{dx}\text{bei}_0(x)$$

If the keyword *Derivative* is set and $|x| > 119$, NaN is returned.

The function KELVIN_BEI0 is based on the work of Burgoyne (1963).

Example

In this example, $\text{bei}_0(0.4)$ and $\text{bei}_0'(0.6)$ are evaluated.

```
PRINT, KELVIN_BEI0(0.4)
     0.0399982
PRINT, KELVIN_BEI0(0.6, /DERIVATIVE)
      0.299798
```

KELVIN_KER0 Function

Evaluates the Kelvin function of the second kind, ker, of order zero.

Usage

result = KELVIN_KER0(*x*)

Input Parameters

x — Argument for which the function value is desired.

Returned Value

result — The value of the Kelvin function of the second kind, ker, of order zero evaluated at *x*.

Input Keywords

Double — If present and nonzero, double precision is used.

Derivative — If present and nonzero, then the derivative of the Kelvin function of the second kind, ker, of order zero evaluated at *x* is computed.

Discussion

The modified Kelvin function $\ker_0(x)$ is defined to be $\Re K_0(xe^{\pi i/4})$. The Bessel function $K_0(x)$ is defined

$$K_0(x) = \int_0^\infty \cos(x \sin t)\, dt$$

If the keyword *Derivative* is set, the function $\ker_0'(x)$ is defined to be

$$\frac{d}{dx}\ker_0(x)$$

If $x < 0$, NaN (Not a Number) is returned. If $x \geq 119$, then zero is returned.

The function KELVIN_KER0 is based on the work of Burgoyne (1963).

Example

In this example, $\ker_0(0.4)$ and $\ker_0'(0.6)$ are evaluated.

```
PRINT,  KELVIN_KER0(0.4)
        1.06262
PRINT,  KELVIN_KER0(0.6, /DERIVATIVE)
       -1.45654
```

KELVIN_KEI0 Function

Evaluates the Kelvin function of the second kind, kei, of order zero.

Usage

result = KELVIN_KEI0(*x*)

Input Parameters

x — Argument for which the function value is desired.

Returned Value

result — The value of the Kelvin function of the second kind, kei, of order zero evaluated at *x*.

Input Keywords

Double — If present and nonzero, double precision is used.

Derivative — If present and nonzero, then the derivative of the Kelvin function of the second kind, kei, of order zero evaluated at *x* is computed.

Discussion

The modified Kelvin function $\text{kei}_0(x)$ is defined to be $\Im K_0(xe^{\pi\imath/4})$. The Bessel function $K_0(x)$ is defined as

$$K_0(x) = \int_0^\infty \cos(x \sin t)\, dt$$

If the keyword *Derivative* is set, the function $\text{kei}_0'(x)$ is defined to be

$$\frac{d}{dx}\text{kei}_0(x)$$

The function KELVIN_KEI0 is based on the work of Burgoyne (1963).

If $x < 0$, NaN (Not a Number) is returned. If $x \geq 119$, zero is returned.

Example

In this example, $\text{kei}_0(0.4)$ and $\text{kei}_0'(0.6)$ are evaluated.

```
PRINT, KELVIN_KEI0(0.4)
     -0.703800
PRINT, KELVIN_KEI0(0.6, /DERIVATIVE)
      0.348164
```

CUM_INTR Function

Evaluates the cumulative interest paid between two periods.

Usage

result = CUM_INTR (*rate, n_periods, present_value, start, end_per, when*)

Input Parameters

rate — Interest rate.

n_periods — Total number of payment periods. *n_periods* cannot be less than or equal to 0.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

start — Starting period in the calculation. *start* cannot be less than 1; or greater than *end_per.*

end_per — Ending period in the calculation.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The cumulative interest paid between the first period and the last period. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function CUM_INTR evaluates the cumulative interest paid between the first period and the last period.

It is computed using the following:

$$\sum_{i=start}^{end\_per} interest_i$$

where $interest_i$ is computed from the function INT_PAYMENT for the i^{th} period.

Example

In this example, CUM_INTR computes the total interest paid for the first year of a 30-year $200,000 loan with an annual interest rate of 7.25%. The payment is made at the end of each month.

```
PRINT, CUM_INTR(0.0725 / 12, 12*30, 200000., 1, 12, 0)
      -14436.5
```

CUM_PRINC Function

Evaluates the cumulative principal paid between two periods.

Usage

result = CUM_PRINC (*rate, n_periods, present_value, start, end_per, when*)

Input Parameters

rate — Interest rate.

n_periods — Total number of payment periods. *n_periods* cannot be less than or equal to 0.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

start — Starting period in the calculation. *start* cannot be less than 1; or greater than *end_per.*

end_per — Ending period in the calculation.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The cumulative principal paid between the first period and the last period. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function CUM_PRINC evaluates the cumulative principal paid between the first period and the last period.

It is computed using the following:

$$\sum_{i=start}^{end\_per} principal_i$$

where $principal_i$ is computed from the function PRINC_PAYMENT for the ith period.

Example

In this example, CUM_PRINC computes the total principal paid for the first year of a 30-year $200,000 loan with an annual interest rate of 7.25%. The payment is made at the end of each month.

```
PRINT, CUM_PRINC(0.0725 / 12, 12*30, 200000., 1, 12, 0)
      -1935.73
```

DEPRECIATION_DB Function

Evaluates the depreciation of an asset using the fixed-declining balance method.

Usage

result = DEPRECIATION_DB (*cost, salvage, life, period, month*)

Input Parameters

cost — Initial value of the asset.

salvage — The value of an asset at the end of its depreciation period.

life — Number of periods over which the asset is being depreciated.

period — Period for which the depreciation is to be computed. *period* cannot be less than or equal to 0, and cannot be greater than *life* +1.

month — Number of months in the first year. *month* cannot be greater than 12 or less than 1.

Returned Value

result — The depreciation of an asset for a specified period using the fixed-declining balance method. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DEPRECIATION_DB computes the depreciation of an asset for a specified period using the fixed-declining balance method. Function DEPRECIATION_DB varies depending on the specified value for the argument *period*, see table below.

Period	Formula
period = 1	$cost \times rate \times \frac{month}{12}$
period = *life*	$(cost - total\ depreciation\ from\ periods) \times rate \times \frac{12 - month}{12}$
period other than 1 or *life*	$(cost - total\ depreciation\ from\ prior\ periods) \times rate$

where

$$rate = 1 - \left(\frac{salvage}{cost}\right)^{\left(\frac{1}{life}\right)}$$

NOTE: *rate* is rounded to three decimal places.

Example

In this example, DEPRECIATION_DB computes the depreciation of an asset, which costs $2,500 initially, a useful life of 3 periods and a salvage value of $500, for each period.

```
ans = fltarr(4)
life = 3
```

```
cost = 2500
salvage = 500
life = 3
month = 6
for period = 1, life+1 DO $
ans(period-1) = depreciation_db(cost, salvage, life, $
                                period, month)
PM, ans
       518.750
       822.219
       480.998
       140.692
```

DEPRECIATION_DDB Function

Evaluates the depreciation of an asset using the double-declining balance method.

Usage

result = DEPRECIATION_DDB (*cost*, *salvage*, *life*, *period*, *factor*)

Input Parameters

cost — Initial value of the asset.

salvage — The value of an asset at the end of its depreciation period.

life — Number of periods over which the asset is being depreciated.

period — Period for which the depreciation is to be computed. *period* cannot be greater than *life*.

factor — Rate at which the balance declines. *factor* must be positive.

Returned Value

result — The depreciation of an asset using the double-declining balance method for a period specified by the user. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DEPRECIATION_DDB computes the depreciation of an asset using the double-declining balance method for a specified period.

It is computed using the following:

$$\left[cost - salvage\left(total\ depreciation\ from\ prior\ periods\right)\right]\left(\frac{factor}{life}\right)$$

Example

In this example, DEPRECIATION_DDB computes the depreciation of an asset, which costs $2,500 initially, lasts 24 periods and a salvage value of $500, for each period.

```
ans = fltarr(24)
life = 24
cost = 2500
salvage = 500
factor = 2
FOR period = 1, life DO $
ans(period-1) = depreciation_ddb(cost, salvage, life, $
                                 period, factor)
PM, ans
       208.333
       190.972
       175.058
       160.470
       147.097
       134.839
       123.603
       113.302
       103.860
       95.2054
```

```
87.2716
79.9990
73.3324
67.2214
61.6196
56.4846
51.7776
47.4628
22.0906
0.00000
0.00000
0.00000
0.00000
0.00000
```

DEPRECIATION_SLN Function

Evaluates the depreciation of an asset using the straight-line method.

Usage

result = DEPRECIATION_SLN (*cost, salvage, life*)

Input Parameters

cost — Initial value of the asset.

salvage — The value of an asset at the end of its depreciation period.

life — Number of periods over which the asset is being depreciated.

Returned Value

result — The straight line depreciation of an asset for its life. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DEPRECIATION_SLN computes the straight line depreciation of an asset for its life.

It is computed using the following:

(*cost-salvage*)*/life*

Example

In this example, DEPRECIATION_SLN computes the depreciation of an asset, which costs $2,500 initially, lasts 24 periods and a salvage value of $500.

```
PRINT, DEPRECIATION_SLN(2500, 500, 24)
       83.3333
```

DEPRECIATION_SYD Function

Evaluates the depreciation of an asset using the sum-of-years digits method.

Usage

result = DEPRECIATION_SYD (*cost*, *salvage*, *life*, *period*)

Input Parameters

cost — Initial value of the asset.

salvage — The value of an asset at the end of its depreciation period.

life — Number of periods over which the asset is being depreciated.

period — Period for which the depreciation is to be computed. *period* cannot be greater than *life*.

Returned Value

result — The sum-of-years digits depreciation of an asset for a specified period. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DEPRECIATION_SYD computes the sum-of-years digits depreciation of an asset for a specified period.

It is computed using the following:

$$(cost - salvage)(period)\ \frac{(life+1)(life)}{2}$$

Example

In this example, DEPRECIATION_SYD computes the depreciation of an asset, which costs $25,000 initially, lasts 15 years, and a salvage value of $5,000, for the 14^{th} year.

```
PRINT, DEPRECIATION_SYD(25000, 5000, 15, 14)
       333.333
```

DEPRECIATION_VDB Function

Evaluates the depreciation of an asset for any given period using the variable-declining balance method.

Usage

result = DEPRECIATION_VDB (*cost*, *salvage*, *life*, *start*, *end_per*, *factor*, *sln*)

Input Parameters

cost — Initial value of the asset.

salvage — The value of an asset at the end of its depreciation period.

life— Number of periods over which the asset is being depreciated.

start — Starting period in the calculation. *start* cannot be less than 1; or greater than *end_per.*

end_per — Final period for the calculation. *end_per* cannot be greater than *life*.

factor — Rate at which the balance declines. *factor* must be positive.

sln — If equal to zero, do not switch to straight-line depreciation even when the depreciation is greater than the declining balance calculation.

Returned Value

result — The depreciation of an asset for any given period, including partial periods, using the variable-declining balance method. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DEPRECIATION_VDB computes the depreciation of an asset for any given period using the variable-declining balance method using the following:

If $sln = 0$

$$\sum_{i=start+1}^{end\_per} ddb_i$$

If $sln \neq 0$

$$A + \sum_{i=k}^{end\_per} \frac{cost - A - salvage}{end - k + 1}$$

where ddb_i is computed from the function DEPRECIATION_DDB for the *i*th period. k = the first period where straight-line depreciation is greater than

$$A = \sum_{i=start+1}^{k-1} ddb_i$$

the depreciation using the double-declining balance method.

Example

In this example, DEPRECIATION_VDB computes the depreciation of an asset between the 10^{th} and 15^{th} year, which costs $25,000 initially, lasts 15 years, and has a salvage value of $5,000.

```
PRINT, DEPRECIATION_VDB(25000., 5000., 15, 10, 15, 2, 0)
        976.69
```

DOLLAR_DECIMAL Function

Converts a fractional price to a decimal price.

Usage

result = DOLLAR_DECIMAL (*fractional_num*, *fraction*)

Input Parameters

fractional_num — Whole number of dollars plus the numerator, as the fractional part.

fraction — Denominator of the fractional dollar. *fraction* must be positive.

Returned Value

result — The dollar price expressed as a decimal number. The dollar price is the whole number part of fractional-dollar plus its decimal part divided by fraction. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DOLLAR_DECIMAL converts a dollar price, expressed as a fraction, into a dollar price, expressed as a decimal number.

It is computed using the following:

$$idollar + \left[fractional\_num - idollar \right] * \frac{10^{(ifrac+1)}}{fraction}$$

where *idollar* is the integer part of *fractional_num*, and *ifrac* is the integer part of *log*(*fraction*).

Example

In this example, DOLLAR_DECIMAL converts $1 1/4 to $1.25.

```
PRINT, DOLLAR_DECIMAL(1.1, 4)
       1.25000
```

DOLLAR_FRACTION Function

Converts a decimal price to a fractional price.

Usage

result = DOLLAR_FRACTION (*decimal_dollar, fraction*)

Input Parameters

decimal_dollar — Dollar price expressed as a decimal number.

fraction — Denominator of the fractional dollar. *fraction* must be positive.

Returned Value

result — The dollar price expressed as a fraction. The numerator is the decimal part of the returned value. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DOLLAR_FRACTION converts a dollar price, expressed as a decimal number, into a dollar price, expressed as a fractional price. If no result can be computed, NaN is returned.

It can be found by solving the following

$$idollar + \frac{\left[decimal\_dollar - idollar\right]}{10^{(ifrac+1)} \,/\, fraction}$$

where *idollar* is the integer part of the *decimal_dollar*, and *ifrac* is the integer part of *log*(*fraction*).

Example

In this example, DOLLAR_FRACTION converts $1.25 to $1 1/4.

```
PRINT, DOLLAR_FRACTION(1.25, 4)
       1.10000
```

EFFECTIVE_RATE Function

Evaluates the effective annual interest rate.

Usage

result = EFFECTIVE_RATE (*nominal_rate, n_periods*)

Input Parameters

nominal_rate — The interest rate as stated on the face of a security.

n_periods — Number of compounding periods per year.

Returned Value

result — The effective annual interest rate. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function EFFECTIVE_RATE computes the continuously-compounded interest rate equivalent to a given periodically-compounded interest rate. The nominal interest rate is the periodically-compounded interest rate as stated on the face of a security.

It can found by solving the following:

$$\left(1+\frac{nominal\_rate}{n\_periods}\right)^{(n\_periods)}-1$$

Example

In this example, EFFECTIVE_RATE computes the effective annual interest rate of the nominal interest rate, 6%, compounded quarterly.

```
PRINT, EFFECTIVE_RATE(0.06, 4)
     0.0613635
```

FUTURE_VALUE Function

Evaluates the future value of an investment.

Usage

result = FUTURE_VALUE (*rate*, *n_periods*, *payment*, *present_value*, *when*)

Input Parameters

rate — Interest rate.

n_periods— Total number of payment periods.

payment — Payment made in each period.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The future value of an investment. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function FUTURE_VALUE computes the future value of an investment. The future value is the value, at some time in the future, of a current amount and a stream of payments.

It can be found by solving the following:

If $rate = 0$

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If $rate \neq 0$

$$present\_value(1+rate)^{n\_periods} + payment\left[1+rate(when)\right]\frac{(1+rate)^{n\_periods}-1}{rate} + future\_value = 0$$

Example

In this example, FUTURE_VALUE computes the value of $30,000 payment made annually at the beginning of each year for the next 20 years with an annual interest rate of 5%.

```
PRINT, FUTURE_VALUE(0.05, 20, -30000.00, -30000.00, 1)
  1.12118e+06
```

FUTURE_VAL_SCHD Function

Evaluates the future value of an initial principal taking into consideration a schedule of compound interest rates.

Usage

result = FUTURE_VAL_SCHD (*principal, schedule*)

Input Parameters

principal — Principal or present value.

schedule — One-dimensional array of interest rates to apply.

Returned Value

result — The future value of an initial principal after applying a schedule of compound interest rates. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function FUTURE_VAL_SCHD computes the future value of an initial principal after applying a schedule of compound interest rates.

It is computed using the following with *count* = N_ELEMENTS (*schedule*):

$$\sum_{i=1}^{count} (principal * schedule_i)$$

where $schedule_i$ = interest rate at the *i*th period.

Example

In this example, FUTURE_VAL_SCHD computes the value of a $10,000 investment after 5 years with interest rates of 5%, 5.1%, 5.2%, 5.3% and 5.4%, respectively.

```
principal = 10000.0
schedule = [ .050, .051, .052, .053, .054 ]
PRINT, FUTURE_VAL_SCHD(principal, schedule)
       12884.8
```

INT_PAYMENT Function

Evaluates the interest payment for an investment for a given period.

Usage

result = INT_PAYMENT (*rate, period, n_periods, present_value, future_value, when*)

Input Parameters

rate — Interest rate.

period— Payment period.

n_periods — Total number of periods.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

future_value — The value, at some time in the future, of a current amount and a stream of payments.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The interest payment for an investment for a given period. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function INT_PAYMENT computes the interest payment for an investment for a given period.

It is computed using the following:

$$\left\{ present\_value(1+rate)^{n\_periods-1} + payment(1+rate*when)\left[\frac{(1+rate)^{n\_periods-1}}{rate}\right]\right\} rate$$

Example

In this example, INT_PAYMENT computes the interest payment for the second year of a 25-year $100,000 loan with an annual interest rate of 8%. The payment is made at the end of each period.

```
PRINT, INT_PAYMENT(0.08, 2, 25, 100000.00, 0.0, 0)
      -7890.57
```

INT_RATE_ANNUITY Function

Evaluates the interest rate per period of an annuity.

Usage

result = INT_RATE_ANNUITY (*n_periods, payment, present_value, future_value, when*)

Input Parameters

n_periods — Total number of periods.

payment — Payment made each period.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

future_value — The value, at some time in the future, of a current amount and a stream of payments.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The interest rate per period of an annuity. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Xguess — If present, the value is used as the initial guess at the interest rate.

Highest — If present, the value is used as the maximum value of the interest rate allowed.

Discussion

Function INT_RATE_ANNUITY computes the interest rate per period of an annuity. An annuity is a security that pays a fixed amount at equally spaced intervals.

It can be found by solving the following:

If $rate = 0$

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If $rate \neq 0$

$$present\_value(1+rate)^{n\_periods} + payment\left[1+rate(when)\right]\frac{(1+rate)^{n\_periods}-1}{rate} + future\_value = 0$$

Example

In this example, INT_RATE_ANNUITY computes the interest rate of a $20,000 loan that requires 70 payments of $350 to pay off the loan.

```
PRINT, 12*INT_RATE_ANNUITY(70, -350, 20000, 0, 1)
     0.0734513
```

INT_RATE_RETURN Function

Evaluates the internal rate of return for a schedule of cash flows.

Usage

result = INT_RATE_RETURN (*values*)

Input Parameters

values — One-dimensional array of cash flows which occur at regular intervals, which includes the initial investment.

Returned Value

result — The internal rate of return for a schedule of cash flows. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Xguess — If present, the value is used as the initial guess at the internal rate of return.

Highest — If present, the value is used as the maximum value of the internal rate of return allowed.

Discussion

Function INT_RATE_RETURN computes the internal rate of return for a schedule of cash flows. The internal rate of return is the interest rate such that a stream of payments has a net present value of zero.

It is found by solving the following with *count* = N_ELEMENTS (*values*):

$$0 = \sum_{i=1}^{count} \frac{value_i}{(1+rate)^i}$$

where $value_i$ = the *i*th cash flow, *rate* is the internal rate of return.

Example

In this example, INT_RATE_RETURN computes the internal rate of return for nine cash flows, $-800, $800, $800, $600, $600, $800, $800, $700 and $3,000, with an initial investment of $4,500.

```
values = [ -4500., -800., 800., 800., 600.,  $
             600., 800., 800., 700., 3000. ]
PRINT, INT_RATE_RETURN(values)
     0.0720820
```

INT_RATE_SCHD Function

Evaluates the internal rate of return for a schedule of cash flows. It is not necessary that the cash flows be periodic.

Usage

result = INT_RATE_SCHD (*values, dates*)

Input Parameters

values — One-dimensional array of cash flows, which includes the initial investment.

dates — One-dimensional array of dates cash flows are made. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

Returned Value

result — The internal rate of return for a schedule of cash flows that is not necessarily periodic. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Xguess — If present, the value is used as the initial guess at the internal rate of return.

Highest — If present, the value is used as the maximum value of the internal rate of return allowed.

Discussion

Function INT_RATE_SCHD computes the internal rate of return for a schedule of cash flows that is not necessarily periodic. The internal rate such that the stream of payments has a net present value of zero.

It can be found by solving the following with *count* = N_ELEMENTS (*values*):

$$0 = \sum_{i=1}^{count} \frac{value_i}{(1+rate)^{\frac{d_i - d_1}{365}}}$$

In the equation above, d_i represents the *i*th payment date. d_1 represents the 1st payment date. $value_i$ represents the *i*th cash flow. *rate* is the internal rate of return.

Example

In this example, INT_RATE_SCHD computes the internal rate of return for nine cash flows, $-800, $800, $800, $600, $600, $800, $800, $700 and $3,000, with an initial investment of $4,500.

```
years = [1998, 1998, 1999, 2000, 2001, 2002, 2003, 2004, $
         2005, 2006]

months = [1, 10, 5, 5, 6, 7, 8, 9, 10, 11]

days = [1, 1, 5, 5, 1, 1, 30, 15, 15, 1]

dates = VAR_TO_DT(years, months, days)

v = [-4500., -800, 800, 800., 600., 600, 800, 800, 700, 3000]

PRINT, INT_RATE_SCHD(v, dates)

     0.0769003
```

MOD_INTERN_RATE Function

Evaluates the modified internal rate of return for a schedule of periodic cash flows.

Usage

result = MOD_INTERN_RATE (v*alues, finance_rate, reinvest_rate*)

Input Parameters

values — One-dimensional array of cash flows.

finance_rate — Interest paid on the money borrowed.

reinvest_rate — Interest rate received on the cash flows.

Returned Value

result — The modified internal rate of return for a schedule of periodic cash flows. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function MOD_INTERN_RATE computes the modified internal rate of return for a schedule of periodic cash flows. The modified internal rate of return differs from the ordinary internal rate of return in assuming that the cash flows are reinvested at the cost of capital, not at the internal rate of return.

It also eliminates the multiple rates of return problem.

It is computed using the following:

$$\left\{\left[\frac{-(pnpv)(1+reinvest\_rate)^{n\_periods}}{(nnpv)(1+finance\_rate)}\right]^{\frac{1}{n\_periods-1}}\right\}-1$$

where *pnpv* is calculated from the function NET_PRES_VALUE for positive values in *values* using *reinvest_rate*, and where *nnpv* is calculated from the function NET_PRES_VALUE for negative values in *values* using *finance_rate*.

Example

In this example, MOD_INTERN_RATE computes the modified internal rate of return for an investment of $4,500 with cash flows of $-800, $800, $800, $600, $600, $800, $800, $700 and $3,000 for 9 years.

```
value = [ -4500., -800., 800., 800., 600., 600., 800., $
          800., 700., 3000. ]
finance_rate = .08
reinvest_rate = .055
PRINT, MOD_INTERN_RATE(value, finance_rate, reinvest_rate)
     0.0665972
```

NET_PRES_VALUE Function

Evaluates the net present value of a stream of unequal periodic cash flows, which are subject to a given discount rate.

Usage

result = NET_PRES_VALUE (*rate*, *values*)

Input Parameters

rate — Interest rate per period.

values — One-dimensional array of equally-spaced cash flows.

Returned Value

result — The net present value of an investment. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function NET_PRES_VALUE computes the net present value of an investment. Net present value is the current value of a stream of payments, after discounting the payments using some interest rate.

It is found by solving the following with *count* = N_ELEMENTS (*values*):

$$\sum_{i=1}^{count} \frac{value_i}{(1+rate)^i}$$

where $value_i$ = the *i*th cash flow.

Example

In this example, NET_PRES_VALUE computes the net present value of a $10 million prize paid in 20 years ($50,000 per year) with an annual interest rate of 6%.

```
rate = 0.06
value = FLTARR(20)
value(*) = 500000.
PRINT, NET_PRES_VALUE(rate, value)
   5.73496e+06
```

NOMINAL_RATE Function

Evaluates the nominal annual interest rate.

Usage

result = NOMINAL_RATE (*effective_rate, n_periods*)

Input Parameters

effective_rate — The amount of interest that would be charged if the interest was paid in a single lump sum at the end of the loan.

n_periods — Number of compounding periods per year.

Returned Value

result — The nominal annual interest rate. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function NOMINAL_RATE computes the nominal annual interest rate. The nominal interest rate is the interest rate as stated on the face of a security.

It is computed using the following:

$$\left[\left(1+effective\_rate\right)^{\frac{1}{n\_periods}}-1\right]*n\_periods$$

Example

In this example, NOMINAL_RATE computes the nominal annual interest rate of the effective interest rate, 6.14%, compounded quarterly.

```
PRINT, NOMINAL_RATE(0.0614, 4)
```

```
0.0600348
```

NUM_PERIODS Function

Evaluates the number of periods for an investment for which periodic and constant payments are made and the interest rate is constant.

Usage

result = NUM_PERIODS (*rate*, *payment*, *present_value*, *future_value*, *when*)

Input Parameters

rate — Interest rate on the investment.

payment — Payment made on the investment.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

future_value — The value, at some time in the future, of a current amount and a stream of payments.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The number of periods for an investment.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function NUM_PERIODS computes the number of periods for an investment based on periodic, constant payment and a constant interest rate.

It can be found by solving the following:

If $rate = 0$

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If $rate \neq 0$

$$present\_value(1+rate)^{n\_periods} + payment\left[1+rate(when)\right]\frac{(1+rate)^{n\_periods}-1}{rate} + future\_value = 0$$

Example

In this example, NUM_PERIODS computes the number of periods needed to pay off a $20,000 loan with a monthly payment of $350 and an annual interest rate of 7.25%. The payment is made at the beginning of each period.

```
PRINT, NUM_PERIODS(0.0725 / 12, -350., 20000., 0., 1)
           70
```

PAYMENT Function

Evaluates the periodic payment for an investment.

Usage

result = PAYMENT (*rate, n_periods, present_value, future_value, when*)

Input Parameters

rate — Interest rate.

n_periods — Total number of periods.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

future_value — The value, at some time in the future, of a current amount and a stream of payments.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The periodic payment for an investment. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function PAYMENT computes the periodic payment for an investment.

It can be found by solving the following:

If $rate = 0$

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If $rate \neq 0$

$$present\_value(1+rate)^{n\_periods} + payment\left[1+rate(when)\right]\frac{(1+rate)^{n\_periods}-1}{rate} + future\_value = 0$$

Example

In this example, PAYMENT computes the periodic payment of a 25-year $100,000 loan with an annual interest rate of 8%. The payment is made at the end of each period.

```
PRINT, PAYMENT(0.08, 25, 100000., 0., 0)
      -9367.88
```

PRESENT_VALUE Function

Evaluates the net present value of a stream of equal periodic cash flows, which are subject to a given discount rate..

Usage

result = PRESENT_VALUE (*rate, n_periods, payment, future_value, when*)

Input Parameters

rate — Interest rate.

n_periods — Total number of periods.

payment — Payment made in each period.

future_value — The value, at some time in the future, of a current amount and a stream of payments.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The present value of an investment. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function PRESENT_VALUE computes the present value of an investment.

If $rate = 0$

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If $rate \neq 0$

$$present\_value(1+rate)^{n\_periods} + payment\left[1+rate(when)\right]\frac{(1+rate)^{n\_periods}-1}{rate} + future\_value = 0$$

Example

In this example, PRESENT_VALUE computes the present value of 20 payments of $500,000 per payment ($10 million) with an annual interest rate of 6%. The payment is made at the end of each period.

```
PRINT, PRESENT_VALUE(0.06, 20, 500000., 0., 0)
-5.73496e+06
```

PRES_VAL_SCHD Function

Evaluates the present value for a schedule of cash flows. It is not necessary that the cash flows be periodic.

Usage

result = PRES_VAL_SCHD (*rate*, *values*, *dates*)

Input Parameters

rate — Interest rate.

values — One-dimensional array of cash flows.

dates — One-dimensional array of dates cash flows are made. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

Returned Value

result — The present value for a schedule of cash flows that is not necessarily periodic. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function PRES_VAL_SCHD computes the present value for a schedule of cash flows that is not necessarily periodic.

It can be found by solving the following with *count* = N_ELEMENTS (*values*):

$$\sum_{i=1}^{count} \frac{value_i}{(1+rate)^{(d_i - d_1)/365}}$$

In the equation above, d_i represents the *i*th payment date, d_1 represents the 1st payment date, and $value_I$ represents the *i*th cash flow.

Example

In this example, PRES_VAL_SCHD computes the present value of 3 payments, $1,000, $2,000 and $1,000, with an interest rate of 5% made on January 3, 1997, January 3, 1999 and January 3, 2000.

```
rate = 0.05
values = [1000.0, 2000.0, 1000.0]
dates = VAR_TO_DT([1997, 1999, 2000], [1, 1, 1], [3, 3, 3])
PRINT, PRES_VAL_SCHD(rate, values, dates)
       3677.90
```

PRINC_PAYMENT Function

Evaluates the payment on the principal for a specified period.

Usage

result = PRINC_PAYMENT (*rate, period, n_periods, present_value, future_value, when*)

Input Parameters

rate — Interest rate.

period — Payment period.

n_periods — Total number of periods.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

future_value — The value, at some time in the future, of a current amount and a stream of payments.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result— The payment on the principal for a given period. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function PRINC_PAYMENT computes the payment on the principal for a given period.

It is computed using the following:

$$payment_i - interest_i$$

where $payment_i$ is computed from the function PAYMENT for the ith period, $interest_i$ is calculated from the function INT_PAYMENT for the ith period.

Example

In this example, PRINC_PAYMENT computes the principal paid for the first year on a 30-year $100,000 loan with an annual interest rate of 8%. The payment is made at the end of each year.

```
PRINT, PRINC_PAYMENT(0.08, 1, 30, 100000., 0., 0)
      -882.742
```

ACCR_INT_MAT Function

Evaluates the interest which has accrued on a security that pays interest at maturity.

Usage

result = ACCR_INT_MAT (*issue, maturity, coupon_rate, par_value, basis*)

Input Parameters

issue — The date on which interest starts accruing. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

coupon_rate — Annual interest rate set forth on the face of the security; the coupon rate.

par_value — Nominal or face value of the security used to calculate interest payments.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	PDay count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The interest which has accrued on a security that pays interest at maturity. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function ACCR_INT_MAT computes the accrued interest for a security that pays interest at maturity:

$$(par\_value)(rate)\left(\frac{A}{D}\right)$$

In the above equation, *A* represents the number of days starting at issue date to maturity date and *D* represents the annual basis.

Example

In this example, ACCR_INT_MAT computes the accrued interest for a security that pays interest at maturity using the US (NASD) 30/360 day count method. The security has a par value of $1,000, the issue date of October 1, 2000, the maturity date of November 3, 2000, and a coupon rate of 6%.

```
issue = VAR_TO_DT(2000, 10, 1)
maturity = VAR_TO_DT(2000, 11, 3)
rate = .06
par = 1000.
basis = 1
PRINT, ACCR_INT_MAT(issue, maturity, rate, par, basis)
       5.33333
```

ACCR_INT_PER Function

Evaluates the interest which has accrued on a security that pays interest periodically.

Usage

result = ACCR_INT_PER (*issue, first_coupon, settlement, coupon_rate, par_value, frequency, basis*)

Input Parameters

issue — The date on which interest starts accruing. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

first_coupon — First date on which an interest payment is due on the security (e.g. coupon date). For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

coupon_rate — Annual interest rate set forth on the face of the security; the coupon rate.

par_value — Nominal or face value of the security used to calculate interest payments.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	PDay count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The accrued interest for a security that pays periodic interest. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function ACCR_INT_PER computes the accrued interest for a security that pays periodic interest.

In the equation below, A_i represents the number days which have accrued for the *i*th quasi-coupon period within the odd period. (The quasi-coupon periods are periods obtained by extending the series of equal payment periods to before or after the actual payment periods.) *NC* represents the number of quasi-coupon

periods within the odd period, rounded to the next highest integer. (The odd period is a period between payments that differs from the usual equally spaced periods at which payments are made.) NL_i represents the length of the normal *i*th quasi-coupon period within the odd period. NL_I is expressed in days.

Function ACCR_INT_PER can be found by solving the following:

$$(par\_value)\left(\frac{rate}{frequency}\left[\sum_{i=}^{NC}\left(\frac{A_i}{NL_i}\right)\right]\right)$$

Example

In this example, ACCR_INT_PER computes the accrued interest for a security that pays periodic interest using the US (NASD) 30/360 day count method. The security has a par value of $1,000, the issue date of October 1, 1999, the settlement date of November 3, 1999, the first coupon date of March 31, 2000, and a coupon rate of 6%.

```
issue = VAR_TO_DT(1999, 10, 1)
first_coupon = VAR_TO_DT(2000, 3, 31)
settlement = VAR_TO_DT(1999, 11, 3)
rate = .06
par = 1000.
frequency = 2
basis = 1
PRINT, ACCR_INT_PER(issue, first_coupon, settlement, $
                    rate, par, frequency, basis)
       5.33333
```

BOND_EQV_YIELD Function

Evaluates the bond-equivalent yield of a Treasury bill.

Usage

result = BOND_EQV_YIELD (*settlement, maturity, discount_rate*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

discount_rate — The interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

Returned Value

result — The bond-equivalent yield of a Treasury bill. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function BOND_EQV_YIELD computes the bond-equivalent yield for a Treasury bill.

It is computed using the following:

if DSM <=182

$$\frac{365 * discount\_rate}{360 - discount\_rate * DSM}$$

otherwise,

$$\frac{-\frac{DSM}{365} + \sqrt{\left(\frac{DSM}{365}\right)^2 - \left(2 * \frac{DSM}{365} - 1\right) * \frac{discount\_rate * DSM}{discount\_rate * DSM - 360}}}{\frac{DSM}{365} - 0.5}$$

In the above equation, *DSM* represents the number of days starting at settlement date to maturity date.

Example

In this example, BOND_EQV_YIELD computes the bond-equivalent yield for a Treasury bill with the settlement date of July 1, 1999, the maturity date of July 1, 2000, and discount rate of 5% at the issue date.

```
PRINT, BOND_EQV_YIELD(settlement, maturity, discount)
     0.052857
```

CONVEXITY Function

Evaluates the convexity for a security.

Usage

result = CONVEXITY (*settlement, maturity, coupon_rate, yield, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

coupon_rate — Annual interest rate set forth on the face of the security; the coupon rate.

yield — Annual yield of the security.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The convexity for a security. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function CONVEXITY computes the convexity for a security. Convexity is the sensitivity of the duration of a security to changes in yield.

It is computed using the following:

$$\frac{1}{(q* frequency)^2}\left\{\sum_{t=1}^{n} t(t+1)\left(\frac{coupon\_rate}{frequency}\right)q^{-t} + n(n+1)q^{-n}\right\}$$
$$\left(\sum_{t=1}^{n}\left(\frac{coupon\_rate}{frequency}\right)q^{-t} + q^{-n}\right)$$

where *n* is calculated from the function COUPON_NUM and

$$q = 1 + \frac{yield}{frequency}.$$

Example

In this example, CONVEXITY computes the convexity for a security with the settlement date of July 1, 1990, and maturity date of July 1, 2000, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1990, 7, 1)
maturity = VAR_TO_DT(2000, 7, 1)
coupon = .075
yield = .09
frequency = 2
basis = 3
PRINT, CONVEXITY(settlement, maturity, $
                 coupon, yield, frequency, basis)
       59.4050
```

COUPON_DAYS Function

Evaluates the number of days in the coupon period containing the settlement date.

Usage

result = COUPON_DAYS (*settlement, maturity, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The number of days in the coupon period which contains the settlement date. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function COUPON_DAYS computes the number of days in the coupon period that contains the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

Example

In this example, COUPON_DAYS computes the number of days in the coupon period of a bond with the settlement date of November 11, 1996, and the maturity date of March 1, 2009, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1996, 11, 11)
maturity = VAR_TO_DT(2009, 3, 1)
frequency =  2
basis =  3
PRINT, COUPON_DAYS(settlement, maturity, frequency, basis)
       182.500
```

COUPON_NUM Function

Evaluates the number of coupons payable between the settlement date and the maturity date.

Usage

result = COUPON_NUM (*settlement, maturity, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The number of coupons payable between the settlement date and the maturity date.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function COUPON_NUM computes the number of coupons payable between the settlement date and the maturity date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

Example

In this example, COUPON_NUM computes the number of coupons payable with the settlement date of November 11, 1996, and the maturity date of March 1, 2009, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1996, 11, 11)
maturity = VAR_TO_DT(2009, 3, 1)
frequency = 2
basis = 3
```

```
PRINT, COUPON_NUM(settlement, maturity, frequency, basis)
           25
```

SETTLEMENT_DB Function

Evaluates the number of days starting with the beginning of the coupon period and ending with the settlement date.

Usage

result = SETTLEMENT_DB (*settlement, maturity, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual

basis	Day count basis
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The number of days in the period starting with the beginning of the coupon period and ending with the settlement date.

Input Keywords

double — If present and nonzero, double precision is used.

Discussion

Function SETTLEMENT_DB computes the number of days from the beginning of the coupon period to the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

Example

In this example, SETTLEMENT_DB computes the number of days from the beginning of the coupon period to November 11, 1996, of a bond with the maturity date of March 1, 2009, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1996, 11, 11)
maturity = VAR_TO_DT(2009, 3, 1)
frequency = 2
basis = 3
PRINT, SETTLEMENT_DB(settlement, maturity, frequency, basis)
           71
```

COUPON_DNC Function

Evaluates the number of days starting with the settlement date and ending with the next coupon date.

Usage

result = COUPON_DNC (*settlement, maturity, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360

basis	Day count basis
3	Actual/365
4	European 30/360

Returned Value

result — The number of days starting with the settlement date and ending with the next coupon date.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function COUPON_DNC computes the number of days from the settlement date to the next coupon date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pp. 17-35.

Example

In this example, COUPON_DNC computes the number of days from November 11, 1996, to the next coupon date of a bond with the maturity date of March 1, 2009, using the Actual/365 day count method.

```
settlement =  VAR_TO_DT(1996, 11, 11)
maturity = VAR_TO_DT(2009, 3, 1)
frequency = 2
basis = 3
PRINT, COUPON_DNC(settlement, maturity, frequency, basis)
          110
```

DEPREC_AMORDEGRC Function

Evaluates the depreciation for each accounting period. During the evaluation of the function a depreciation coefficient based on the asset life is applied.

Usage

result = DEPREC_AMORDEGRC (*cost, issue, first_period, salvage, period, rate, basis*)

Input Parameters

cost— Initial value of the asset.

issue— The date on which interest starts accruing. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

first_period— Date of the end of the first period. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

salvage — The value of an asset at the end of its depreciation period.

period — Depreciation for the accounting period to be computed.

rate — Depreciation rate.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The depreciation for each accounting period. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DEPREC_AMORDEGRC computes the depreciation for each accounting period. This function is similar to DEPREC_AMORLINC. However, in this function a depreciation coefficient based on the asset life is applied during the evaluation of the function.

Example

In this example, DEPREC_AMORDEGRC computes the depreciation for the second accounting period using the US (NASD) 30/360 day count method. The security has the issue date of November 1, 1999, end of first period of November 30, 2000, cost of $2,400, salvage value of $300, depreciation rate of 15%.

```
issue = VAR_TO_DT(1999, 11, 1)

first_period = VAR_TO_DT(2000, 11, 30)

cost = 2400.

salvage = 300.

period = 2

rate = .15

basis = 1

PRINT, DEPREC_AMORDEGRC(cost, issue, first_period, $
                        salvage, period, rate, basis

       335.000
```

DEPREC_AMORLINC Function

Evaluates the depreciation for each accounting period. This function is similar to DEPREC_AMORDEGRC, except that DEPREC_AMORDEGRC has a

depreciation coefficient that is applied during the evaluation that is based on the asset life.

Usage

result = DEPREC_AMORLINC (*cost, issue, first_period, salvage, period, rate, basis*)

Input Parameters

cost — Initial value of the asset.

issue — The date on which interest starts accruing. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

first_period— Date of the end of the first period. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

salavge — The value of an asset at the end of its depreciation period.

period— Depreciation for the accounting period to be computed.

rate — Depreciation rate.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The depreciation for each accounting period. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DEPREC_AMORLINC computes the depreciation for each accounting period.

Example

In this example, DEPREC_AMORLINC computes the depreciation for the second accounting period using the US (NASD) 30/360 day count method. The security has the issue date of November 1, 1999, end of first period of November 30, 2000, cost of $2,400, salvage value of $300, depreciation rate of 15%.

```
issue = VAR_TO_DT(1999, 11, 1)

first_period = VAR_TO_DT(2000, 11, 30)

cost = 2400.

salvage = 300.

period = 2

rate = .15

basis = 1

PRINT, DEPREC_AMORLINC(cost, issue, first_period, $
                       salvage, period, rate, basis)

       360.000
```

DISCOUNT_PR Function

Evaluates the price of a security sold for less than its face value.

Usage

result = DISCOUNT_PR (*settlement, maturity, discount_rate, redemption, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

discount_rate — The interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

redemption — Redemption value per $100 face value of the security.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The price per face value for a discounted security. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DISCOUNT_PR computes the price per $100 face value of a discounted security.

It is computed using the following:

$$redemption - (discount\_rate)\left[redemption\left(\frac{DSM}{B}\right)\right]$$

In the equation above, *DSM* represents the number of days starting at the settlement date and ending with the maturity date. *B* represents the number of days in a year based on the annual basis.

Example

In this example, DISCOUNT_PR computes the price of the discounted bond with the settlement date of July 1, 2000, and maturity date of July 1, 2001, at the discount rate of 5% using the US (NASD) 30/360 day count method.

```
settlement = VAR_TO_DT(2000, 7, 1)
maturity = VAR_TO_DT(2001, 7, 1)
discount = .05
redemption = 100.
basis = 1
PRINT, DISCOUNT_PR(settlement, maturity, discount, $
                   redemption, basis
       95.0000
```

DISCOUNT_RT Function

Evaluates the interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

Usage

result = DISCOUNT_RT (*settlement, maturity, price, redemption, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

price — Price per $100 face value of the security.

redemption — Redemption value per $100 face value of the security.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The discount rate for a security. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DISCOUNT_RT computes the discount rate for a security. The discount rate is the interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

It is computed using the following:

$$\left(\frac{redemption - price}{price}\right)\left(\frac{B}{DSM}\right)$$

In the equation above, *B* represents the number of days in a year based on the annual basis and *DSM* represents the number of days starting with the settlement date and ending with the maturity date.

Example

In this example, DISCOUNT_RT computes the discount rate of a security which is selling at $97.975 with the settlement date of February 15, 2000, and maturity date of June 10, 2000, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(2000, 2, 15)

maturity = VAR_TO_DT(2000, 6, 10)

price = 97.975

redemption = 100.

basis = 3

PRINT, DISCOUNT_RT(settlement, maturity, price,  $
                   redemption, basis)

     0.0637177
```

DISCOUNT_YLD Function

Evaluates the annual yield of a discounted security.

Usage

result = DISCOUNT_YLD (*settlement, maturity, price, redemption, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

price — Price per $100 face value of the security.

redemption — Redemption value per $100 face value of the security.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The annual yield for a discounted security. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DISCOUNT_YLD computes the annual yield for a discounted security.

It is computed using the following:

$$\left(\frac{redemption - price}{price}\right)\left(\frac{B}{DSM}\right)$$

In the equation above, *B* represents the number of days in a year based on the annual basis, and *DSM* represents the number of days starting with the settlement date and ending with the maturity date.

Example

In this example, DISCOUNT_YLD computes the annual yield for a discounted security which is selling at $95.40663 with the settlement date of July 1, 1995,

and maturity date of July 1, 2005, using the US (NASD) 30/360 day count method.

```
settlement = VAR_TO_DT(1995, 7, 1)

maturity = VAR_TO_DT(2005, 7, 1)

price = 95.40663

redemption = 105.

basis = 1

PRINT, DISCOUNT_YLD(settlement, maturity, price, redemption$
                    basis)
```

```
0.0100552
```

DURATION Function

Evaluates the annual duration of a security where the security has periodic interest payments.

Usage

result = DURATION (*settlement, maturity, coupon_rate, yield, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

coupon_rate — Annual interest rate set forth on the face of the security; the coupon rate.

yield — Annual yield of the security.

frequency— Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The annual duration of a security with periodic interest payments. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DURATION computes the Maccaluey's duration of a security with periodic interest payments. The Maccaluey's duration is the weighted-average time to the payments, where the weights are the present value of the payments.

It is computed using the following:

$$\left(\frac{\frac{DSC}{E} * 100}{\left(1+\frac{yield}{freq}\right)^{\left(N-1+\frac{DSC}{E}\right)}} + \sum_{k=1}^{N} \left(\left(\frac{100 * coupon\_rate}{freq * \left(1+\frac{yield}{freq}\right)^{\left(k-1+\frac{DSC}{E}\right)}} \right) * \left(k-1+\frac{DSC}{E} \right) \right) \right) * \frac{1}{freq}$$

$$\frac{100}{\left(1+\frac{yield}{freq}\right)^{N-1+\frac{DSC}{E}}} + \sum_{k=1}^{N} \left(\frac{100 * coupon\_rate}{freq * \left(1+\frac{yield}{freq}\right)^{k-1+\frac{DSC}{E}}} \right)$$

In the equation above, *DSC* represents the number of days starting with the settlement date and ending with the next coupon date. *E* represents the number of days within the coupon period. *N* represents the number of coupons payable from the settlement date to the maturity date. *freq* represents the frequency of the coupon payments annually.

Example

In this example, DURATION computes the annual duration of a security with the settlement date of July 1, 1995, and maturity date of July 1, 2005, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1995, 7, 1)

maturity = VAR_TO_DT(2005, 7, 1)

coupon = .075

yield = .09

frequency = 2

basis = 3

PRINT, DURATION(settlement, maturity, coupon,  $
                yield, frequency, basis)

       7.04195
```

INT_RATE_SEC Function

Evaluates the interest rate of a fully invested security.

Usage

result = INT_RATE_SEC (*settlement, maturity, investment, redemption, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

investment — The total amount one has invested in the security.

redemption — Amount to be received at maturity.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The interest rate for a fully invested security. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function INT_RATE_SEC computes the interest rate for a fully invested security.

It is computed using the following:

$$\left(\frac{redemption - investment}{investment}\right)\left(\frac{B}{DSM}\right)$$

In the equation above, *B* represents the number of days in a year based on the annual basis, and *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date.

Example

In this example, INT_RATE_SEC computes the interest rate of a $7,000 investment with the settlement date of July 1, 1995, and maturity date of July 1, 2005, using the Actual/365 day count method. The total amount received at the end of the investment is $10,000.

```
settlement = VAR_TO_DT(1995, 7, 1)

maturity = VAR_TO_DT(2005, 7, 1)

investment = 7000.

redemption = 10000.

basis = 3

PRINT, INT_RATE_SEC(settlement, maturity, investment,$
                    redemption, basis)

     0.0428219
```

DURATION_MAC Function

Evaluates the modified Macauley duration of a security.

Usage

result = DURATION_MAC (*settlement, maturity, coupon_rate, yield, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

coupon_rate — Annual interest rate set forth on the face of the security; the coupon rate.

yield — Annual yield of the security.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The modified Macauley duration of a security is returned. The security has an assumed par value of $100. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DURATION_MAC computes the modified Macauley duration for a security with an assumed par value of $100.

It is computed using the following:

$$\frac{duration}{1+\left(\frac{yield}{frequency}\right)}$$

where *duration* is calculated from the function DURATION.

Example

In this example, DURATION_MAC computes the modified Macauley duration of a security with the settlement date of July 1, 1995, and maturity date of July 1, 2005, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1995, 7, 1)

maturity = VAR_TO_DT(2005, 7, 1)

coupon = .075

yield = .09

frequency = 2

basis = 3

PRINT, DURATION_MAC(settlement, maturity, $
                    coupon, yield, frequency, basis)

       6.73871
```

COUPON_NCD Function

Evaluates the first coupon date which follows the settlement date.

Usage

result = COUPON_NCD (*settlement, maturity, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The first coupon date which follows the settlement date.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function COUPON_NCD computes the next coupon date after the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol 1, pages 17-35.

Example

In this example, COUPON_NCD computes the next coupon date of a bond with the settlement date of November 11, 1996, and the maturity date of March 1, 2009, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1996, 11, 11)
maturity = VAR_TO_DT(2009, 3, 1)
frequency = 2
basis = 3
ans = COUPON_NCD(settlement, maturity, frequency, basis)
DT_TO_STR, ans, d, Date_Fmt=4
01/March/1997
```

COUPON_PCD Function

Evaluates the coupon date which immediately precedes the settlement date.

Usage

result = COUPON_PCD (*settlement, maturity, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

fequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The coupon date which immediately precedes the settlement date.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function COUPON_PCD computes the coupon date which immediately precedes the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol 1, pages 17-35.

Example

In this example, COUPON_PCD computes the previous coupon date of a bond with the settlement date of November 11, 1986, and the maturity date of March 1, 1999, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1996, 11, 11)
maturity = VAR_TO_DT(2009, 3, 1)
frequency = 2
basis = 3
ans = COUPON_PCD(settlement, maturity, frequency, basis)
DT_TO_STR, ans, d, Date_Fmt=4
PRINT, d
01/September/1996
```

PRICE_PERIODIC Function

Evaluates the price, per $100 face value, of a security that pays periodic interest.

Usage

result = PRICE_PERIODIC (*settlement, maturity, rate, yield, redemption, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid.

rate — Annual interest rate set forth on the face of the security; the coupon rate.

yield — Annual yield of the security.

redemption — Redemption value per $100 face value of the security.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The price per $100 face value of a security that pays periodic interest. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function PRICE_PERIODIC computes the price per $100 face value of a security that pays periodic interest.

It is computed using the following:

$$\left(\frac{redemption}{\left(1+\frac{yield}{frequency}\right)^{\left(N-1+\frac{DSC}{E}\right)}}\right)+\left[\sum_{k=1}^{N}\frac{100*\frac{rate}{frequency}}{\left(1+\frac{yield}{frequency}\right)^{\left(k-1+\frac{DSC}{E}\right)}}\right]-\left(100*\frac{rate}{frequency}*\frac{A}{E}\right)$$

In the above equation, *DSC* represents the number of days in the period starting with the settlement date and ending with the next coupon date. *E* represents the number of days within the coupon period. *N* represents the number of coupons payable in the timeframe from the settlement date to the redemption date. *A* represents the number of days in the timeframe starting with the beginning of coupon period and ending with the settlement date.

Example

In this example, PRICE_PERIODIC computes the price of a bond that pays coupon every six months with the settlement of July 1, 1995, the maturity date of July 1, 2005, a annual rate of 6%, annual yield of 7% and redemption value of $105 using the US (NASD) 30/360 day count method.

```
settlement = VAR_TO_DT(1995, 7, 1)
maturity = VAR_TO_DT(2005, 7, 1)
rate = .06
yield = .07
redemption = 105.
frequency = 2
basis = 1
PRINT, PRICE_PERIODIC(settlement, maturity, rate, yield, $
                      redemption, frequency, basis)
       95.4067
```

PRICE_MATURITY Function

Evaluates the price, per $100 face value, of a security that pays interest at maturity.

Usage

result = PRICE_MATURITY (*settlement, maturity, issue, rate, yield, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

issue — The date on which interest starts accruing. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

rate — Annual interest rate set forth on the face of the security; the coupon rate.

yield — Annual yield of the security.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The price per $100 face value of a security that pays interest at maturity. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function PRICE_MATURITY computes the price per $100 face value of a security that pays interest at maturity.

It is computed using the following:

$$\left[\frac{100+\left(\frac{DIM}{B}*rate*100\right)}{1+\left(\frac{DSM}{B}*yield\right)}\right]-\left(\frac{A}{B}*rate*100\right)$$

In the equation above, *B* represents the number of days in a year based on the annual basis. *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date. *DIM* represents the number of days in the period starting with the issue date and ending with the maturity date. *A* represents the number of days in the period starting with the issue date and ending with the settlement date.

Example

In this example, PRICE_MATURITY computes the price at maturity of a security with the settlement date of August 1, 2000, maturity date of July 1, 2001 and issue date of July 1, 2000, using the US (NASD) 30/360 day count method. The security has 5% annual yield and 5% interest rate at the date of issue.

```
settlement = VAR_TO_DT(2000, 8, 1)
maturity = VAR_TO_DT(2001, 7, 1)
issue = VAR_TO_DT(2000, 7, 1)
rate = .05
yield = .05
basis = 1
```

```
PRINT, PRICE_MATURITY(settlement, maturity, issue, $
                      rate, yield, basis)
       99.9817
```

MATURITY_REC Function

Evaluates the amount one receives when a fully invested security reaches the maturity date.

Usage

result = MATURITY_REC (*settlement, maturity, investment, discount_rate, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

investment — The total amount one has invested in the security.

discount_rate — The interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The amount one receives when a fully invested security reaches its maturity date. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function MATURITY_REC computes the amount received at maturity for a fully invested security.

It is computed using the following:

$$\frac{investment}{1-\left(discount\_rate * \frac{DIM}{B}\right)}$$

In the equation above, *B* represents the number of days in a year based on the annual basis, and *DIM* represents the number of days in the period starting with the issue date and ending with the maturity date.

Example

In this example, MATURITY_REC computes the amount received of a $7,000 investment with the settlement date of July 1, 1995, maturity date of July 1, 2005 and discount rate of 6%, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1995, 7, 1)

maturity = VAR_TO_DT(2005, 7, 1)

investment = 7000.

discount = .06

basis =  3

PRINT, MATURITY_REC(settlement, maturity, investment,$
                    discount, basis)

       17521.6
```

TBILL_PRICE Function

Evaluates the price per $100 face value of a Treasury bill.

Usage

result = TBILL_PRICE (*settlement, maturity, discount_rate*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

discount_rate — The interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

Returned Value

result — The price per $100 face value of a Treasury bill. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function TBILL_PRICE computes the price per $100 face value for a Treasury bill.

It is computed using the following:

$$100\left(1-\frac{discount\_rate * DSM}{360}\right)$$

In the equation above, *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date (any maturity date that is more than one calendar year after the settlement date is excluded).

Example

In this example, TBILL_PRICE computes the price for a Treasury bill with the settlement date of July 1, 2000, the maturity date of July 1, 2001, and a discount rate of 5% at the issue date.

```
settlement = VAR_TO_DT(2000, 7, 1)
maturity = VAR_TO_DT(2001, 7, 1)
discount = .05
PRINT, TBILL_PRICE(settlement, maturity, discount)
       94.9306
```

TBILL_YIELD Function

Evaluates the yield of a Treasury bill.

Usage

result = TBILL_YIELD (*settlement, maturity, price*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

price — Price per $100 face value of the Treasury bill.

Returned Value

result — The yield for a Treasury bill. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function TBILL_YIELD computes the yield for a Treasury bill.

It is computed using the following:

$$\left(\frac{100 - price}{price}\right)\left(\frac{360}{DSM}\right)$$

In the equation above, *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date (any maturity date that is more than one calendar year after the settlement date is excluded).

Example

In this example, TBILL_YIELD computes the yield for a Treasury bill with the settlement date of July 1, 2000, the maturity date of July 1, 2001, and priced at $94.93.

```
settlement = VAR_TO_DT(2000, 7, 1)
maturity = VAR_TO_DT(2001, 7, 1)
price = 94.93
PRINT, TBILL_YIELD(settlement, maturity, price)
     0.0526762
```

YEAR_FRACTION Function

Evaluates the fraction of a year represented by the number of whole days between two dates.

Usage

result = YEAR_FRACTION (*date_start*, *date_end*, *basis*)

Input Parameters

date_start — Initial date. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

date_end — Ending date. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The fraction of a year represented by the number of whole days between two dates. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function YEAR_FRACTION computes the fraction of the year.

It is computed using the following:

$$A / D$$

where A = the number of days from *start* to *end*, D = annual basis.

Example

In this example, YEAR_FRACTION computes the year fraction between August 1, 2000, and July 1, 2001, using the US (NASD) 30/360 day count method.

```
date_start = VAR_TO_DT(2000, 8, 1)
date_end = VAR_TO_DT(2001, 7, 1)
basis = 1
PRINT, YEAR_FRACTION(date_start, date_end, basis)
      0.916667
```

YIELD_MATURITY Function

Evaluates the annual yield of a security that pays interest at maturity.

Usage

result = YIELD_MATURITY (*settlement, maturity, issue, rate, price, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

issue — The date on which interest starts accruing. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

rate — Interest rate at date of issue of the security.

price — Price per $100 face value of the security.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The annual yield of a security that pays interest at maturity. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function YIELD_MATURITY computes the annual yield of a security that pays interest at maturity.

It is computed using the following:

$$\left\{\frac{\left[\left[1+\left(\frac{DIM}{B}*rate\right)\right]-\left[\frac{price}{100}+\left(\frac{A}{B}*rate\right)\right]\right]}{\frac{price}{100}+\left(\frac{A}{B}*rate\right)}\right\}*\left(\frac{B}{DSM}\right)$$

In the equation above, *DIM* represents the number of days in the period starting with the issue date and ending with the maturity date. *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date. *A* represents the number of days in the period starting with the issue date and ending with the settlement date. *B* represents the number of days in a year based on the annual basis.

Example

In this example, YIELD_MATURITY computes the annual yield of a security that pays interest at maturity which is selling at $95.40663 with the settlement date of August 1, 2000, the issue date of July 1, 2000, the maturity date of July 1, 2010, and the interest rate of 6% at the issue using the US (NASD) 30/360 day count method.

```
settlement = VAR_TO_DT(2000, 8, 1)

maturity = VAR_TO_DT(2010, 7, 1)

issue = VAR_TO_DT(2000, 7, 1)

rate = .06

price = 95.40663

basis = 1

PRINT, YIELD_MATURITY(settlement, maturity, issue, $
                      rate, price, basis)

     0.0673905
```

YIELD_PERIODIC Function

Evaluates the yield of a security that pays periodic interest.

Usage

result = YIELD_PERIODIC (*settlement, maturity, coupon_rate, price, redemption, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

coupon_rate — Annual coupon rate.

price — Price per $100 face value of the security.

redemption — Redemption value per $100 face value of the security.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The yield of a security that pays interest periodically. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Xguess — If present, the value is used as the initial guess at the internal rate of return.

Highest — If present, the value is used as the maximum value of the internal rate of return allowed.

Discussion

Function YIELD_PERIODIC computes the yield of a security that pays periodic interest. If there is one coupon period use the following:

$$\left\{\frac{\left(\frac{redemption}{100}+\frac{coupon\_rate}{frequency}\right)-\left[\frac{price}{100}+\left(\frac{A}{E}*\frac{coupon\_rate}{frequency}\right)\right]}{\frac{price}{100}+\left(\frac{A}{E}*\frac{coupon\_rate}{frequency}\right)}\right\}\left(\frac{frequency*E}{DSR}\right)$$

In the equation above, *DSR* represents the number of days in the period starting with the settlement date and ending with the redemption date. *E* represents the number of days within the coupon period. *A* represents the number of days in the period starting with the beginning of coupon period and ending with the settlement date.

If there is more than one coupon period use the following:

$$price - \left(\left(\frac{redemption}{\left(1 + \frac{yield}{frequency}\right)^{\left(N - 1 + \frac{DSC}{E}\right)}} \right) + \left[\sum_{k=1}^{N} \frac{100 * \frac{rate}{frequency}}{\left(1 + \frac{yield}{frequency}\right)^{\left(k - 1 + \frac{DSC}{E}\right)}} \right] - \left(100 * \frac{rate}{frequency} * \frac{A}{E} \right) \right) = 0$$

In the equation above, *DSC* represents the number of days in the period from the settlement to the next coupon date. *E* represents the number of days within the coupon period. *N* represents the number of coupons payable in the period starting with the settlement date and ending with the redemption date. *A* represents the number of days in the period starting with the beginning of the coupon period and ending with the settlement date.

Example

In this example, YIELD_PERIODIC computes yield of a security which is selling at $95.40663 with the settlement date of July 1, 1985, the maturity date of July 1, 1995, and the coupon rate of 6% at the issue using the US (NASD) 30/360 day count method.

```
settlement = VAR_TO_DT(2000, 7, 1)
maturity = VAR_TO_DT(2010, 7, 1)
coupon_rate = .06
price = 95.40663
redemption = 105.
frequency = 2
basis = 1
PRINT, YIELD_PERIODIC(settlement, maturity, coupon_rate, $
                      price, redemption, frequency, basis)
     0.0700047
```

10

CHAPTER

Basic Statistics and Random Number Generation

Contents of Chapter

Introduction

The functions for computations of basic statistics generally have relatively simple input parameters. The data are input in either a one- or two-dimensional array. As usual, when a two-dimensional array is used, the rows contain observations and the columns represent variables. Most of the functions in this chapter allow for missing values. Missing value codes can be set by using function MACHINE.

Several functions in this chapter perform statistical tests. These functions generally return a "*p*-value" for the test, often as the return value for the C function. The *p*-value is between 0 and 1 and is the probability of observing data that would yield a test statistic as extreme or more extreme under the assumption of the null hypothesis. Hence, a small *p*-value is evidence for the rejection of the null hypothesis.

Overview of Random Number Generation

The "Random Numbers" section describes functions for the generation of random numbers and of random samples and permutations. These functions are useful for applications in Monte Carlo or simulation studies. Before using any of the random-number generators, the generator must be initialized by selecting a *seed* or starting value. This can be done by calling the *Set* keyword with the RANDOMOPT procedure. If the user does not select a seed, one is generated using the system clock. A seed needs to be selected only once in a program, unless two or more separate streams of random numbers are maintained. There are other utility functions in this chapter for selecting the form of the basic generator, restarting simulations, and maintaining separate simulation streams.

In the following discussions, the phrases "random numbers," "random deviates," "deviates," and "variates" are used interchangeably. The phrase "pseudorandom" is sometimes used to emphasize that the numbers generated are really not "random" since they result from a deterministic process. The usefulness of pseudorandom numbers is derived from the similarity, in a statistical sense, of samples of the pseudorandom numbers to samples of observations from the specified distributions. In short, while the pseudorandom numbers are completely deterministic and repeatable, they simulate the realizations of independent and identically distributed random variables.

Basic Uniform Generator

The random-number generators in this chapter use a multiplicative congruential method. The form of the generator is as follows:

$$x_i = cx_{i-1} \bmod (2^{31} - 1)$$

Each x_i is then scaled into the unit interval (0,1). If the multiplier, c, is a primitive root modulo $2^{31} - 1$ (which is a prime), then the generator has a maximal period of $2^{31} - 2$. However, there are several other considerations. See Knuth (1981) for a general discussion. The possible values for c in the generators are 16807, 397204094, and 950706376. The selection is made by using the *Gen_Option* keyword with the RANDOMOPT procedure. The choice of 16807 results in the fastest execution time, but other evidence suggests that the performance of 950706376 is best among these three choices (Fishman and Moore 1982). If no selection is made explicitly, the functions use the multiplier 16807, which has been in use for some time (Lewis et al. 1969).

The default action of the RANDOM function is the generation of uniform (0,1) numbers. This function is portable in the sense that, given the same seed, it produces the same sequence in all computer/compiler environments.

Shuffled Generators

The user also can select a shuffled version of these generators using the *Gen_Option* keyword with the RANDOMOPT procedure. The shuffled generators use a scheme due to Learmonth and Lewis (1973). In this scheme, a table is filled with the first 128 uniform (0,1) numbers resulting from the simple multiplicative congruential generator. Then, for each x_i from the simple generator, the low-order bits of x_i are used to select a random integer, j, from 1 to 128. The j-th entry in the table is then delivered as the random number, and x_i, after being scaled into the unit interval, is inserted into the j-th position in the table. This scheme is similar to that of Bays and Durham (1976), and their analysis is applicable to this scheme as well.

Setting the Seed

Using the RANDOMOPT procedure with the *Set* keyword, the seed of the generator can be set and can be retrieved with the *Get* keyword. Prior to invoking any generator in this section, the user can call RANDOMOPT to initialize the seed, which is an integer variable with a value between 1 and 2147483647. If it is not initialized by RANDOMOPT, a random seed is obtained from the system clock. Once it is initialized, the seed need not be set again.

If the user wants to restart a simulation, RANDOMOPT can be used to obtain the final seed value of one run to be used as the starting value in a subsequent run.

CHISQTEST Function

Performs a chi-squared goodness-of-fit test.

Usage

result = CHISQTEST(*f*, *n_categories*, *x*)

Input Parameters

f — Scalar string specifying a user-supplied function. Function *f* accepts one scalar parameter and returns the hypothesized, cumulative distribution function at that point.

n_categories — Number of cells into which the observations are to be tallied.

x — One-dimensional array containing the vector of data elements for this test.

Returned Value

result — The *p*-value for the goodness-of-fit chi-squared statistic.

Input Keywords

Double — If present and nonzero, double precision is used.

N_Params_Estimated — Number of parameters estimated in computing the cumulative distribution function.

Equal_Cutpoints — If present and nonzero, equal probability cutpoints are used. Keyword *Equal_Cutpoints* should not be used if *Cutpoints* is present.

Cutpoints — Specifies the named variable containing user-defined cutpoints to be used by CHISQTEST. Keywords *Cutpoints* and *Equal_Cutpoints* cannot be used together.

Frequencies — Named variable into which the array containing the vector frequencies for the observations stored in *x* is stored.

Lower_Bound — Lower bound of the range of the distribution. If *Lower Bound = Upper Bound*, a range on the whole real line is used (the default). If the lower and upper endpoints are different, points outside of the range of these bounds are ignored. Distributions conditional on a range can be specified when *Lower_Bound* and *Upper_Bound* are used. If *Lower_Bound* is specified, then *Upper_Bound* also must be specified. By convention, *Lower_Bound* is excluded from the first interval, but *Upper_Bound* is included in the last interval.

Upper_Bound — Upper bound of the range of the distribution. If *Lower Bound = Upper Bound*, a range on the whole real line is used (the default). If the lower and upper endpoints are different, points outside of the range of these bounds are ignored. Distributions conditional on a range can be specified when *Lower_Bound* and *Upper_Bound* are used. If *Upper_Bound* is specified, then *Lower_Bound* also must be specified. By convention, *Lower_Bound* is excluded from the first interval, but *Upper_Bound* is included in the last interval.

Output Keywords

Used_Cutpoints — Specifies the named variable into which the cutpoints to be used by CHISQTEST are stored.

Chi_Squared — Named variable into which the chi-squared test statistic is stored.

Df — Named variable into which the degrees of freedom for the chi-squared goodness-of-fit test are stored.

Cell_Counts — Named variable into which the cell counts are stored. The cell counts are the observed frequencies in each of the *n_categories* cells.

Cell_Expected — Named variable into which the cell expected values are stored. The expected value of a cell is the expected count in the cell given that the hypothesized distribution is correct.

Cell_Chisq — Named variable into which an array of length *n_categories* containing the cell contributions to chi-squared is stored.

Discussion

Function CHISQTEST performs a chi-squared goodness-of-fit test that a random sample of observations is distributed according to a specified theoretical cumulative distribution. The theoretical distribution, which may be continuous, discrete, or a mixture of discrete and continuous distributions, is specified by

the user-defined function *f*. Because the user is allowed to give a range for the observations, a test that is conditional upon the specified range is performed.

Parameter *n_categories* gives the number of intervals into which the observations are to be divided. By default, equiprobable intervals are computed by CHISQTEST, but intervals that are not equiprobable can be specified (through the use of keyword *Cutpoints*).

Regardless of the method used to obtain the cutpoints, the intervals are such that the lower endpoint is not included in the interval, while the upper endpoint is always included. If the cumulative distribution function has discrete elements, then user-provided cutpoints should always be used since CHISQTEST cannot determine the discrete elements in discrete distributions.

By default, the lower and upper endpoints of the first and last intervals are *–infinity* and *+infinity*. The endpoints can be specified by using the keywords *Lower_Bound* and *Upper_Bound*.

A tally of counts is maintained for the observations in *x* as follows:

- If the cutpoints are specified by the user, the tally is made in the interval to which x_i belongs using the endpoints specified by the user.
- If the cutpoints are determined by CHISQTEST, then the cumulative probability at x_i, $F(x_i)$, is computed by the function *f*.

The tally for x_i is made in interval number

$$\lfloor mF(x_i) + 1 \rfloor,$$

where m = *n_categories* and

$$\lfloor \cdot \rfloor$$

is the function that takes the greatest integer that is no larger than the parameter of the function. Thus, if the computer time required to calculate the cumulative distribution function is large, user-specified cutpoints may be preferred in order to reduce the total computing time.

If the expected count in any cell is less than 1, then a rule of thumb is that the chi-squared approximation may be suspect. A warning message to this effect is issued in this case, as well as when an expected value is less than 5.

Programming Notes

The user must supply a function *f* with calling sequence *F*(*y*) that returns the value of the cumulative distribution function at any point *y* in the (optionally) specified range.

Many of the cumulative distribution functions in the *PV-WAVE: IMSL Statistics Reference* can be used for *f*. It is, however, necessary to write a user-defined PV-WAVE:IMSL Mathematics function that calls the CDF, and then pass the name of this user-defined function for *f*.

Example

This example illustrates the use of CHISQTEST on a randomly generated sample from the normal distribution. One-thousand randomly generated observations are tallied into 10 equiprobable intervals. In this example, the null hypothesis is not rejected.

```
.RUN
    ; Define the hypothesized, cumulative distribution function.


- FUNCTION user_cdf, k
-   RETURN, NORMALCDF(k)
- END
RANDOMOPT, Set = 123457
x = RANDOM(1000, /Normal)
    ; Generate normal deviates.
p_value = CHISQTEST("user_cdf", 10, x)
    ; Perform chi-squared test.
PM, p_value
    ; Output the results.
0.154603
```

Warning Errors

STAT_EXPECTED_VAL_LESS_THAN_1 — An expected value is less than 1.

STAT_EXPECTED_VAL_LESS_THAN_5 — An expected value is less than 5.

Fatal Errors

STAT_ALL_OBSERVATIONS_MISSING — All observations contain missing values.

STAT_INCORRECT_CDF_1 — Function *f* is not a cumulative distribution function. The value at the lower bound must be nonnegative, and the value at the upper bound must not be greater than 1.

STAT_INCORRECT_CDF_2 — Function *f* is not a cumulative distribution function. The probability of the range of the distribution is not positive.

STAT_INCORRECT_CDF_3 — Function *f* is not a cumulative distribution function. Its evaluation at an element in *x* is inconsistent with either the evaluation at the lower or upper bound.

STAT_INCORRECT_CDF_4 — Function *f* is not a cumulative distribution function. Its evaluation at a cutpoint is inconsistent with either the evaluation at the lower or upper bound.

STAT_INCORRECT_CDF_5 — An error has occurred when inverting the cumulative distribution function. This function must be continuous and defined over the whole real line.

FREQTABLE Function

Tallies observations into a one-way frequency table.

Usage

result = FREQTABLE(*x*, *nbins*)

Input Parameters

x — One-dimensional array containing the observations.

nbins — Number of intervals (bins).

Returned Value

result — One-dimensional array containing the counts.

Input Keywords

Double — If present and nonzero, double precision is used.

Lower_Bound — Used with *Upper_Bound* to specify two semi-infinite intervals that are used as the initial and last interval. The initial interval is closed on the right and includes *Lower_Bound* as its right endpoint. The last interval is open on the left and includes all values greater than *Upper_Bound*. The remaining *nbins* – 2 intervals are of length

(*Upper_Bound* – *Lower_Bound*) / (*nbins* – 2)

and are open on the left and closed on the right. The keyword *Upper_Bound* also must be specified with this keyword. Parameter *nbins* must be greater than or equal to 3 for this option.

Upper_Bound — Used along with *Lower_Bound* to specify two semi-infinite intervals that are used as the initial and last interval. The initial interval is closed on the right and includes *Lower_Bound* as its right endpoint. The last interval is open on the left and includes all values greater than *Upper_Bound*. The remaining *nbins* – 2 intervals are of length (*Upper_Bound* – *Lower_Bound*) / (*nbins* – 2) and are open on the left and closed on the right. The keyword *Lower_Bound* must also be specified with this keyword. Parameter *nbins* must be greater than or equal to 3 for this option.

Cutpoints — Specifies a one-dimensional array of length *nbins* containing the cutpoints to use. This option allows unequal intervals. The initial interval is closed on the right and contains the initial cutpoint as its right endpoint. The last interval is open on the left and includes all values greater than the last cutpoint. The remaining *nbins* – 2 intervals are open on the left and closed on the right. Parameter *nbins* must be greater than 3 for this option. If *Cutpoints* is used, then no other keywords should be specified.

Class_Marks — Specifies a one-dimensional array containing equally spaced class marks in ascending order. The class marks are the midpoints of each of the *nbins*, and each interval is taken to have length (*Class_Marks*(1) – *Class_Marks*(0)). Parameter *nbins* must be greater than or equal to 2 for this option. If *Class_Marks* is used, then no other keywords should be specified.

Discussion

The default action of FREQTABLE is to group data into *nbins* categories of size (max (*x*) – min (*x*)) / *nbins*. The initial interval is closed on the left and open on the right. The remaining intervals are open on the left and closed on the right. Using keywords, the types of intervals used may be changed.

If *Upper_Bound* and *Lower_Bound* are specified, two semi-infinite intervals are used as the initial and last interval. The initial interval is closed on the right and includes *Lower_Bound* as its right endpoint. The last interval is open on the left and includes all values greater than *Upper_Bound*. The remaining *nbins* – 2 intervals are of length (*Upper_Bound* – *Lower_Bound*) / (*nbins* – 2) and are open on the left and closed on the right. Parameter *nbins* must be greater than or equal to 3 for this option.

If keyword *Class_Marks* is used, equally spaced class marks in ascending order must be provided in an array of length *nbins*. The class marks are the midpoints of each of the *nbins*, and each interval is taken to have the following length:

(*Class_Marks*(1) – *Class_Marks*(0))

Parameter *nbins* must be greater than or equal to 2 for this option.

If keyword *Cutpoints* is used, cutpoints (bounders) must be provided in an array of length *nbins*. This option allows unequal intervals. The initial interval is closed on the right and contains the initial cutpoint as its right endpoint. The last interval is open on the left and includes all values greater than the last cutpoint. The remaining *nbins* – 2 intervals are open on the left and closed on the right. Parameter *nbins* must be greater than 3 for this option.

Example

The data for this example is from Hinkley (1977) and Velleman and Hoaglin (1981). Data includes measurements (in inches) of precipitation in Minneapolis/St. Paul during the month of March for 30 consecutive years.

```
x = [0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47,$
   1.43, 3.37, 2.20, 3.00, 3.09, 1.51, 2.10,$
   0.52, 1.62, 1.31, 0.32, 0.59, 0.81, 2.81,$
   1.87, 1.18, 1.35, 4.75, 2.48, 0.96, 1.89,$
   0.90, 2.05]
       ; Define the data set.
table = FREQTABLE(x, 10)
   ; Call FREQTABLE with nbins = 10.
PRINT, '    Bin Number   Count' &$
PRINT, '    ----------   -----'  &$
FOR i = 0, 9 DO PRINT, i + 1, table(i)
```

```
Bin Number   Count
----------   -----
 1       4.00000
```

```
 2        8.00000
 3        5.00000
 4        5.00000
 5        3.00000
 6        1.00000
 7        3.00000
 8        0.00000
 9        0.00000
10       1.00000
```

RANKS Function

Computes the ranks, normal scores, or exponential scores for a vector of observations.

Usage

result = RANKS(*x*)

Input Parameters

x — One-dimensional array containing the observations to be ranked.

Returned Value

result — A one-dimensional array containing the rank (or optionally, a transformation of the rank) of each observation.

Input Keywords

Double — If present and nonzero, double precision is used.

Average_Tie, *or*
Highest, *or*
Lowest, *or*
Random_Split — At most, one of these keywords can be set to a nonzero value to change the method used to assign a score to tied observations.

Keyword	Method
Average_Tie	average of the scores of the tied observations (default)
Highest	highest score in the group of ties
Lowest	lowest score in the group of ties
Random_Split	tied observations are randomly split using a random-number generator

Fuzz — Value used to determine when two items are tied. If ABS(x(I) – x(J)) is less than or equal to *Fuzz*, then x(I) and x(J) are said to be tied.

Default: *Fuzz* = 0.0

Ranks, *or*
Blom_Scores, *or*
Tukey_Scores, *or*
Vdw_Scores, *or*
Exp_Norm_Scores, *or*
Savage_Scores — At most, one of these keywords can be set to a nonzero value to specify the type of values returned.

Keyword	Result
Ranks	ranks (default)
Blom_Scores	Blom version of normal scores
Tukey_Scores	Tukey version of normal scores
Vdw_Scores	Van der Waerden version of normal scores
Exp_Norm_Scores	expected value of normal order statistics (for tied observations, the average of the expected normal scores)
Savage_Scores	Savage scores (expected value of exponential order statistics)

Discussion

Ties

If the assignment RANK = RANKS(x) is made, then in data without ties, the output values are the ordinary ranks (or a transformation of the ranks) of the

data in x. If $x(i)$ has the smallest value among the values in x and there is no other element in x with this value, then RANK(i) = 1. If both $x(i)$ and $x(j)$ have the same smallest value, then the output value depends on the option used to break ties.

Keyword	Result
Average_Tie	*result* (i) = *result* (j) = 1.5
Highest	*result* (i) = *result* (j) = 2.0
Lowest	*result* (i) = *result* (j) = 1.0
Random_Split	*result* (i) = 1.0 and *result* (j) = 2.0 or, randomly, *result* (i) = 2.0 and *result* (j) = 1.0

When the ties are resolved randomly, function RANDOM (page 506) is used to generate random numbers. Different results occur from different executions of the program unless the "seed" of the random-number generator is set explicitly by use of the function RANDOMOPT ( ).

Scores

Normal and other functions of the ranks can optionally be returned. Normal scores can be defined as the expected values, or approximations to the expected values, of order statistics from a normal distribution. The simplest approximations are obtained by evaluating the inverse cumulative normal distribution function, NORMALCDF (with keyword *Inverse*), at the ranks scaled into the open interval (0,1).

In the Blom version (Blom 1958), the scaling transformation for the rank r_i ($1 \le r_i \le n$, where n is the sample size) is $(r_i - 3 / 8) / (n + 1 / 4)$. The Blom normal score corresponding to the observation with rank r_i is

$$\Phi^{-1}\left(\frac{r_i - 3/8}{n + 1/4} \right)$$

where $\Phi(\cdot)$ is the normal cumulative distribution function.

Adjustments for ties are made after the normal score transformation; that is, if $x(i)$ equals $x(j)$ (within *Fuzz*) and their value is the k-th smallest in the data set, the Blom normal scores are determined for ranks of k and $k + 1$. Then, these normal scores are averaged or selected in the manner specified. (Whether the

transformations are made first or the ties are resolved first is irrelevant, except when *Average_Tie* is specified.)

In the Tukey version (Tukey 1962), the scaling transformation for the rank r_i is $(r_i - 1 / 3) / (n + 1 / 3)$. The Tukey normal score corresponding to the observation with rank r_i follows:

$$\Phi^{-1}\left(\frac{r_i - 1/3}{n + 1/3} \right)$$

Ties are handled in the same way as for the Blom normal scores.

In the Van der Waerden version (see Lehmann 1975, p. 97), the scaling transformation for the rank r_i is $r_i / (n + 1)$. The Van der Waerden normal score corresponding to the observation with rank r_i is as follows:

$$\Phi^{-1}\left(\frac{r_i}{n + 1} \right)$$

Ties are handled in the same way as for the Blom normal scores.

When option *Exp_Norm_Scores* is nonzero, the output values are the expected values of the normal order statistics from a sample of size n = N_ELEMNTS(x). If the value in $x(i)$ is the k-th smallest, then the value output in RANK (i) is $E(z_k)$, where $E(\cdot)$ is the expectation operator, and z_k is the k-th order statistic in a sample of size n from a standard normal distribution. Ties are handled in the same way as for the Blom normal scores.

Savage scores are the expected values of the exponential order statistics from a sample of size n. These values are called Savage scores because of their use in a test discussed by Savage (1956) and Lehmann (1975). If the value in $x(i)$ is the k-th smallest, then the value output in RANK (i) is $E(y_k)$ where y_k is the k-th order statistic in a sample of size n from a standard exponential distribution. The expected value of the k-th order statistic from an exponential sample of size n follows:

$$\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{n-k+1}$$

Ties are handled in the same way as for the Blom normal scores.

Example

The data for this example, from Hinkley (1977), contains 30 observations. Note that the fourth and sixth observations are tied, and the third and twentieth observations are tied.

```
x = [0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47,$
   1.43, 3.37, 2.20, 3.00, 3.09, 1.51, 2.10,$
   0.52, 1.62, 1.31, 0.32, 0.59, 0.81, 2.81,$
   1.87, 1.18, 1.35, 4.75, 2.48, 0.96, 1.89,$
   0.90, 2.05]

r = RANKS(x)
   ; Call RANKS.

FOR i = 0, 29 DO PM, i + 1, r(i), Format = '(i5, f7.1)'
```

```
    1     5.0
    2    18.0
    3     6.5
    4    11.5
    5    21.0
    6    11.5
    7     2.0
    8    15.0
    9    29.0
   10    24.0
   11    27.0
   12    28.0
   13    16.0
   14    23.0
   15     3.0
   16    17.0
   17    13.0
   18     1.0
   19     4.0
   20     6.5
   21    26.0
   22    19.0
   23    10.0
   24    14.0
   25    30.0
   26    25.0
   27     9.0
   28    20.0
```

```
29      8.0
30     22.0
```

RANDOMOPT Procedure

Uses keywords to set or retrieve the random number seed or to select the form of the IMSL random number generator.

Usage

RANDOMOPT

Input Parameters

Procedure RANDOMOPT does not have any positional Input Parameters. Keywords are required for specific actions to be taken.

Input Keywords

Gen_Option — Indicator of the generator. The random-number generator is a multiplicative, congruential generator with modulus $2^{31} - 1$. Keyword *Gen_Option* is used to choose the multiplier and to determine whether or not shuffling is done.

Gen_Option	Generator
1	multiplier 16807 used (default)
2	multiplier 16807 used with shuffling
3	multiplier 397204094 used
4	multiplier 397204094 used with shuffling
5	multiplier 950706376 used
6	multiplier 950706376 used with shuffling
7	GFSR, with the recursion $X_t = X_{t-1563} \oplus X_{t-96}$ is used

Set — Seed of the random-number generator. The seed must be in the range (0, 2147483646). If the seed is zero, a value is computed using the system

clock; hence, the results of programs using the PV- WAVE:IMSL Statistics random-number generators are different at various times.

Substream_seed — If present and nonzero, then a seed for the congruential generators that do not do shuffling that will generate random numbers beginning 100,000 numbers farther along will be returned in keyword *Get*. If keyword *Substream_seed* is set, then keyword *Get* is required.

Output Keywords

Get — Named variable into which the value of the current random-number seed is stored.

Current_option — Named variable into which the value of the current random-number generator option is stored.

Discussion

Procedure RANDOMOPT is designed to allow a user to set certain key elements of the random-number generator functions.

The uniform pseudorandom-number generators use a multiplicative congruential method, or a generalized feedback shift register. The choice of generator is determined by keyword *Gen_Option*. The chapter introduction and the description of function RANDOM may provide some guidance in the choice of the form of the generator. If no selection is made explicitly, the generators use the multiplier 16807 without shuffling. This form of the generator has been in use for some time (Lewis et al. 1969).

Keyword *Set* is used to initialize the seed used in the PV- WAVE:IMSL Statistics random-number generators. See the chapter introduction for details of the various gererator options. The seed can be reinitialized to a clock-dependent value by calling RANDOMOPT with *Set* set to zero.

A common use of keyword *Set* is in conjunction with the keyword *Get* to restart a simulation. Keyword *Get* retrieves the current value of the "seed" used in the random-number generators.

If keyword *Substream_seed* is set, RANDOMOPT determines another seed, such that if one of the IMSL multiplicative congruential generators, using no shuffling, went through 100,000 generations starting with *Substream_seed*, the next number in that sequence would be the first number in the sequence that begins with the returned seed.

Note that *Substream_seed* works only when a multiplicative congruential generator without shuffling is used. This means that either the routine RANDOMOPT has not been called at all or that it has been last called with *Gen_Option* having a value of 1, 3, or 5.

For many of the IMSL generators for nonuniform distributions that do not use the inverse CDF method, the distance between the sequences generated starting with *Substream_seed* and starting with the returned seed may be less than 100,000. This is because the nonuniform generators that use other techniques may require more than one uniform deviate for each output deviate.

The reason that one may want two seeds that generate sequences a known distance apart is for blocking Monte Carlo experiments or for running parallel streams.

Example 1

This example illustrates the statements required to restart a simulation using the keywords *Get* and *Set*. The example shows that restarting the sequence of random numbers at the value of the last seed generated is the same as generating the random numbers all at once.

```
seed = 123457
nrandom = 5
RANDOMOPT, Set = seed
    ; Set the seed using the keyword Set.
r1 = RANDOM(nrandom)
PM, r1, Title = 'First Group of Random Numbers'
```

```
First Group of Random Numbers
 0.966220
 0.260711
 0.766262
 0.569337
 0.844829
RANDOMOPT, Get = seed
    ; Get the current value of the seed using the keyword Get.
RANDOMOPT, Set = seed
    ; Set the seed.
r2 = RANDOM(nrandom)
PM, r2, $
    Title = 'Second Group of Random Numbers'
Second Group of Random Numbers
```

```
 0.0442665
 0.987184
 0.601350
 0.896375
 0.380854
RANDOMOPT, Set = 123457
    ; Reset the seed to the original seed.
r3 = RANDOM(2 * nrandom)
PM, r3, Title = 'Both Groups of Random Numbers'
Both Groups of Random Numbers
 0.966220
 0.260711
 0.766262
 0.569337
 0.844829
 0.0442665
 0.987184
 0.601350
 0.896375
 0.380854
```

Example 2

In this example, RANDOMOPT is used to determine seeds for 4 separate streams, each 200,000 numbers apart, for a multiplicative congruential generator without shuffling. (Since RANDOMOPT is not invoked to select a generator, the multiplier is 16807.) Since the streams are 200,000 numbers apart, each seed requires two invocations of RANDOMOPT with keyword *Substream_seed*. All of the streams are non-overlapping, since the period of the underlying generator is 2,147,483,646.

```
RANDOMOPT, GEN_OPTION = 1
is1 = 123457;
RANDOMOPT, Get = itmp, Substream_seed = is1
RANDOMOPT, Get = is2, Substream _seed = itmp
RANDOMOPT, Get = itmp, Substream _seed = is2
RANDOMOPT, Get = is3, Substream _seed = itmp
RANDOMOPT, Get = itmp, Substream _seed = is3
RANDOMOPT, Get = is4, Substream _seed = itmp
PRINT, is1, is2, is3, is4
       123457   2016130173     85016329    979156171
```

RANDOM Function

Generates pseudorandom numbers. The default distribution is a uniform (0, 1) distribution, but many different distributions can be specified through the use of keywords.

Usage

result = RANDOM(*n*)

Generally, it is best to first identify the desired distribution from the "*Discussion*" section, then refer to the "*Input Keywords*" section for specific usage instructions.

Input Parameters

n — Number of random numbers to generate.

Returned Value

result — A one-dimensional array of length *n* containing the random numbers. If one of the keywords *Sphere*, *Multinomial*, or *Mvar_Normal* are used, then a two-dimensional array is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Parameters — Specifies parameters for the distribution used by RANDOM to generate numbers. Some distributions require this keyword to execute successfully. The type and range of these parameters depends upon which distribution is specified. See the keyword for the desired distribution or the *Discussion* section for more details.

Beta — If present and nonzero, the random numbers are generated from a beta distribution. Requires the *Parameters* keyword to specify the parameters (*p*, *q*) for the distribution. The parameters *p* and *q* must be positive.

Binomial — If present and nonzero, the random numbers are generated from a binomial distribution. Requires the *Parameters* keyword to specify the parameters (*p*, *n*) for the distribution. The parameter *n* is the number of Bernoulli trials, and it must be greater than zero. The parameter *p* represents the probability of success on each trial, and it must be between 0.0 and 1.0.

Cauchy — If present and nonzero, the random numbers are generated from a Cauchy distribution.

Chi_squared — If present and nonzero, the random numbers are generated from a chi-squared distribution. Requires the *Parameters* keyword to specify the parameter *Df* for the distribution. The parameter *Df* is the number of degrees of freedom for the distribution, and it must be positive.

Discrete_unif — If present and nonzero, the random numbers are generated from a discrete uniform distribution. Requires the *Parameters* keyword to specify the parameter *k* for the distribution. This generates integers in the range from 1 to *k* (inclusive) with equal probability. The parameter *k* must be positive.

Exponential — If present and nonzero, the random numbers are generated from a standard exponential distribution.

Gamma — If present and nonzero, the random numbers are generated from a standard Gamma distribution. Requires the *Parameters* keyword to specify the parameter *a* for the distribution. The parameter *a* is the shape parameter of the distribution, and it must be positive *n*.

Geometric — If present and nonzero, the random numbers are generated from a geometric distribution. Requires the *Parameters* keyword to specify the parameter *P* for the distribution. The parameter *P* must be positive and less than 1.0.

Hypergeometric — If present and nonzero, the random numbers are generated from a hypergeometric distribution. Requires the *Parameters* keyword to specify the parameters (*M*, *N*, *L*) for the distribution. The parameter *N* represents the number of items in the sample, *M* is the number of special items in the population, and *L* is the total number of items in the population. The parameters *N* and *M* must be greater than zero, and *L* must be greater than both *N* and *M*.

Logarithmic — If present and nonzero, the random numbers are generated from a logarithmic distribution. Requires the *Parameters* keyword to specify the parameter *a* for the distribution. The parameter *a* must be greater than zero.

Lognormal — If present and nonzero, the random numbers are generated from a lognormal distribution. Requires the *Parameters* keyword to specify the parameters (μ, σ) for the distribution. The parameter μ is the mean of the distribution, while σ represents the standard deviation.

Mix_Exponential — If present and nonzero, the random numbers are generated from a mixture of two exponential distributions. Requires the *Parameters* keyword to specify the parameters (θ_1, θ_2, *p*) for the distribution. The parameters θ_1 and θ_2 are the means for the two distributions; both must be positive, and θ_1 must be greater than θ_2. The parameter *p* is the relative probability of the

θ_1 distribution, and it must be non-negative and less than or equal to $\theta_1/(\theta_1 - \theta_2)$.

Neg_binomial — If present and nonzero, the random numbers are generated from a negative binomial distribution. Requires the *Parameters* keyword to specify the parameters (*r*, *p*) for the distribution. The parameter *r* must be greater than zero. If *r* is an integer, the generated deviates can be thought of as the number of failures in a sequence of Bernoulli trials before *r* successes occur. The parameter *p* is the probability of success on each trial. It must be greater than the machine epsilon, and less than 1.0.

Normal — If present and nonzero, the random numbers are generated from a standard normal distribution using an inverse CDF method.

Permutation — If present and nonzero, then generate a pseudorandom permutation.

Poisson — If present and nonzero, the random numbers are generated from a Poisson distribution. Requires the *Parameters* keyword to specify the parameter θ for the distribution. The parameter θ represents the mean of the distribution, and it must be positive.

Sample_indices— If present and nonzero, generate a simple pseudorandom sample of indices. Requires the *Parameters* keyword to specify the parameter *npop*, the number of items in the population.

Sphere— If present and nonzero, the random numbers are generated on a unit circle or *K*-dimensional sphere. Requires the *Parameters* keyword to specify the parameter *k*, the dimension of the circle ($k = 2$) or of the sphere.

Stable — If present and nonzero, the random numbers are generated from a stable distribution. Requires the *Parameters* keyword to specify the parameters *A* and *bprime* for the stable distribution. *A* is the characteristic exponent of the stable distribution. *A* must be positive and less than or equal to 2. *bprime* is related to the usual skewness parameter β of the stable distribution.

Student_t — If present and nonzero, the random numbers are generated from a Student's *t* distribution. Requires the *Parameters* keyword to specify the parameter *Df* for the distribution. The *Df* parameter is the number of degrees of freedom for the distribution, and it must be positive.

Triangular — If present and nonzero, the random numbers are generated from a triangular distribution.

Uniform — If present and nonzero, the random numbers are generated from a uniform (0, 1) distribution. The default action of this returns random numbers from a uniform (0, 1) distribution.

Von_mises — If present and nonzero, the random numbers are generated from a von Mises distribution. Requires the *Parameters* keyword to specify the parameter *c* for the function. The parameter *c* must be greater than one-half the machine epsilon.

Weibull — If present and nonzero, the random numbers are generated from a Weibull distribution. Requires the *Parameters* keyword to specify the parameters (*a*, *b*) for the distribution. The parameter *a* is the shape parameter, and it is required. The parameter *b* is the scale parameter, and is optional (Default: *b* = 1.0).

Mvar_Normal — If present and nonzero, the random numbers are generated from a multivariate normal distribution. Keywords *Mvar_Normal* and *Covariances* must be specified to return numbers from a multivariate normal distribution.

Covariances — Two-dimensional, square matrix containing the variance-covariance matrix. The two-dimensional array returned by RANDOM is of the following size:

n by N_ELEMENTS(*Covariances*(*, 0))

Keywords *Mvar_Normal* and *Covariances* must be specified to return numbers from a multivariate normal distribution.

Multinomial — If present and nonzero, the random numbers are generated from a multinomial distribution. Requires the *Parameters* keyword to specify the parameter (*ntrials*) for the distribution, and the keyword *Probabilities* to specify the array containing the probabilities of the possible outcomes. The value if *ntrials* is the multinomial parameter indicating the number of independent trials.

Probabilities — Specifies the array containing the probabilities of the possible outcomes. The elements of *P* must be positive and must sum to 1.0.

Keywords *Multinomial* and *Probabilities* must be specified to return numbers from a *Multinomial* distribution.

NOTE The keywords *A*, *Pin*, *Qin*, and *Theta* are still supported, but are now deprecated. Please use the *Parameters* keyword instead.

Discussion

Function RANDOM is designed to return random numbers from any of a number of different distributions. The determination of which distribution to

generate the random numbers from is based on the presence of a keyword or groups of keywords. If RANDOM is called without any keywords, then random numbers from a uniform (0, 1) distribution are returned.

Uniform (0,1) Distribution

The default action of RANDOM generates pseudorandom numbers from a uniform (0, 1) distribution using a multiplicative, congruential method. The form of the generator follows:

$$x_i \equiv cx_{i-1} \bmod (2^{31} - 1)$$

Each x_i is then scaled into the unit interval (0, 1). The possible values for c in the generators are 16807, 397204094, and 950706376. The selection is made by using the RANDOMOPT procedure with the *Gen_Option* keyword. The choice of 16807 results in the fastest execution time. If no selection is made explicitly, the functions use the multiplier 16807. See RANDOMOPT on page 502 for futher discussion of generator options.

The RANDOMOPT procedure called with the *Set* keyword is used to initialize the seed of the random-number generator.

The user can select a shuffled version of these generators. In this scheme, a table is filled with the first 128 uniform (0, 1) numbers resulting from the simple multiplicative congruential generator. Then, for each x_i from the simple generator, the low-order bits of x_i are used to select a random integer, j, from 1 to 128. The j-th entry in the table is then delivered as the random number, and x_i, after being scaled into the unit interval, is inserted into the j-th position in the table.

The values returned are positive and less than 1.0. Some values returned may be smaller than the smallest relative spacing; however, it may be the case that some value, for example $r(i)$, is such that $1.0 - r(i) = 1.0$.

Deviates from the distribution with uniform density over the interval (a, b) can be obtained by scaling the output. See Example 3 on page 523 for more details.

Normal Distribution

Calling RANDOM with keyword *Normal* generates pseudorandom numbers from a standard normal (Gaussian) distribution using an inverse CDF technique. In this method, a uniform (0,1) random deviate is generated. Then, the inverse of the normal distribution function is evaluated at that point using the NORMALCDF function with keyword *Inverse*.

If the *Parameters* keyword is specified in addition to *Normal*, RANDOM generates pseudorandom numbers using an acceptance/rejection technique due to Kinderman and Ramage (1976). In this method, the normal density is represented as a mixture of densities over which a variety of acceptance/rejection methods due to Marsaglia (1964), Marsaglia and Bray (1964), and Marsaglia et al. (1964) are applied. This method is faster than the inverse CDF technique.

Deviates from the normal distribution with mean specific mean and standard deviation can be obtained by scaling the output from RANDOM. See Example 3 on page 523 for more details.

Exponential Distribution

Calling RANDOM with keyword *Exponential* generates pseudorandom numbers from a standard exponential distribution. The probability density function is $f(x) = e^{-x}$, for $x > 0$. Function RANDOM uses an antithetic inverse CDF technique. In other words, a uniform random deviate U is generated, and the inverse of the exponential cumulative distribution function is evaluated at $1.0 - U$ to yield the exponential deviate.

Poisson Distribution

Calling RANDOM with keywords *Poisson* and *Parameters*= θ generates pseudorandom numbers from a Poisson distribution with positive mean θ. The probability function follows:

$$f(x) = (e^{-\theta}\theta^{x})/x!\ , \quad \text{for } x = 0, 1, 2, \ldots$$

If θ is less than 15, RANDOM uses an inverse CDF method; otherwise, the PTPE method of Schmeiser and Kachitvichyanukul (1981) is used. (See also Schmeiser 1983.) The PTPE method uses a composition of four regions, a triangle, a parallelogram, and two negative exponentials. In each region except the triangle, acceptance/rejection is used. The execution time of the method is essentially insensitive to the mean of the Poisson.

Gamma Distribution

Calling RANDOM with keywords *Gamma* and *Parameters*=*a* generates pseudorandom numbers from a Gamma distribution with shape parameter *a* and unit scale parameter. The probability density function follows:

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x} \qquad \text{for } x \geq 0$$

Various computational algorithms are used depending on the value of the shape parameter *a*. For the special case of $a = 0.5$, squared and halved normal deviates are used; for the special case of $a = 1.0$, exponential deviates are generated. Otherwise, if *a* is less than 1.0, an acceptance-rejection method due to Ahrens, described in Ahrens and Dieter (1974), is used. If *a* is greater than 1.0, a 10-region rejection procedure developed by Schmeiser and Lal (1980) is used.

The Erlang distribution is a standard Gamma distribution with the shape parameter having a value equal to a positive integer; hence, RANDOM generates pseudorandom deviates from an Erlang distribution with no modifications required.

Beta Distribution

Calling RANDOM with keywords *Beta*, and *Parameters*=[*p*,*q*] generates pseudorandom numbers from a beta distribution. With *p* and *q* both positive, the probability density function is

$$f(x) = \frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)} x^{p-1}(1-x)^{q-1}$$

where $\Gamma(\cdot)$ is the Gamma function.

The algorithm used depends on the values of *p* and *q*. Except for the trivial cases of $p = 1$ or $q = 1$, in which the inverse CDF method is used, all the methods use acceptance/rejection. If *p* and *q* are both less than 1, the method of Jöhnk (1964) is used. If either *p* or *q* is less than 1 and the other is greater than 1, the method of Atkinson (1979) is used. If both *p* and *q* are greater than 1, algorithm BB of Cheng (1978), which requires very little setup time, is used if *x* is less than 4, and algorithm B4PE of Schmeiser and Babu (1980) is used if *x* is greater than or equal to 4. Note that for *p* and *q* both greater than 1, calling RANDOM to generate random numbers from a beta distribution a loop getting less than four variates on each call yields the same set of deviates as executing one call and getting all the deviates at once.

The values returned are less than 1.0 and greater than ε, where ε is the smallest positive number such that $1.0 - \varepsilon$ is less than 1.0.

Multivariate Normal Distribution

Calling RANDOM with keywords *Mvar_Normal* and *Covariances* generates pseudorandom numbers from a multivariate normal distribution with mean vector consisting of all zeros and variance-covariance matrix defined using keyword *Covariances*. First, the Cholesky factor of the variance-covariance

matrix is computed. Then, independent random normal deviates with mean zero and variance 1 are generated, and the matrix containing these deviates is post-multiplied by the Cholesky factor. Because the Cholesky factorization is performed in each invocation, it is best to generate as many random vectors as needed at once.

Deviates from a multivariate normal distribution with means other than zero can be generated by using RANDOM with keywords *Mvar_Normal* and *Covariances*, then adding the vectors of means to each row of the result.

Binomial Distribution

Calling RANDOM with keywords *Binomial, Parameters*= [*p, n*] generates pseudorandom numbers from a binomial distribution with parameters *n* and *p*. Parameters *n* and *p* must be positive, and *p* must less than 1. The probability function (where *n* = *Binom_n* and *p* = *Binom_p*) is

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

for $x = 0, 1, 2, \ldots, n$.

The algorithm used depends on the values of *n* and *p*. If $n * p < 10$ or *p* is less than machine epsilon, the inverse CDF technique is used; otherwise, the BTPE algorithm of Kachitvichyanukul and Schmeiser (see Kachitvichyanukul 1982) is used. This is an acceptance /rejection method using a composition of four regions. (TPE=Triangle, Parallelogram, Exponential, left and right.)

Cauchy Distribution

Calling RANDOM with the keyword *Cauchy* generates pseudorandom numbers from a Cauchy distribution. The probability density function is

$$f(x) = \frac{S}{\pi[S^2 + (x-T)^2]}$$

where *T* is the median and $T - S$ is the first quartile. This function first generates standard Cauchy random numbers ($T = 0$ and $S = 1$) using the technique described below, and then scales the values using *T* and *S*.

Use of the inverse CDF technique would yield a Cauchy deviate from a uniform (0, 1) deviate, u, as tan [p (u − 0.5)]. Rather than evaluating a tangent directly,

however, RANDOM generates two uniform (−1, 1) deviates, x_1 and x_2. These values can be thought of as sine and cosine values. If

$$x_1^2 + x_2^2$$

is less than or equal to 1, then x_1/x_2 is delivered as the unscaled Cauchy deviate; otherwise, x_1 and x_2 are rejected and two new uniform (−1, 1) deviates are generated. This method is also equivalent to taking the ration of two independent normal deviates.

Chi-squared Distribution

Calling RANDOM with keywords *Chi_squared* and *Parameters=Df* generates pseudorandom numbers from a chi-squared distribution with *Df* degrees of freedom. If *Df* is an even integer less than 17, the chi-squared deviate *r* is generated as

$$r = -2\ln\left(\prod_{i=1}^{n} u_i\right)$$

where $n = Df/2$ and the u_i are independent random deviates from a uniform (0, 1) distribution. If *Df* is an odd integer less than 17, the chi-squared deviate is generated in the same way, except the square of a normal deviate is added to the expression above. If *Df* is greater than 16 or is not an integer, and if it is not too large to cause overflow in the gamma random number generator, the chi-squared deviate is generated as a special case of a gamma deviate.

Mixed Exponential Distribution

Calling RANDOM with keywords *Mix_Exponential*, and *Parameters*= $[\theta_1, \theta_2]$ generates pseudorandom numbers from a mixture of two exponential distributions. The probability density function is

$$f(x) = \frac{p}{\theta_1} e^{-x/\theta_1} + \frac{1-p}{\theta_2} e^{-x/\theta_2}$$

for $x > 0$.

In the case of a convex mixture, that is, the case $0 < p < 1$, the mixing parameter p is interpretable as a probability; and RANDOM with probability p generates an exponential deviate with mean θ_1, and with probability $1 - p$ gen-

erates an exponential with mean θ_2. When p is greater than 1, but less than $\theta_1/(\theta_1 - \theta_2)$, then either an exponential deviate with mean θ_1 or the sum of two exponentials with means θ_1 and θ_2 is generated. The probabilities are $q = p - (p - 1)(\theta_1/\theta_2)$ and $1 - q$, respectively, for the single exponential and the sum of the two exponentials.

Geometric Distribution

Calling RANDOM with keywords *Geometric* and *Parameters=P* generates pseudorandom numbers from a geometric distribution. The parameter *P* is the probability of getting a success on any trial. A geometric deviate can be interpreted as the number of trials until the first success (including the trial in which the first success is obtained). The probability function is

$$f(x) = P(1 - P)^{x-1}$$

for $x = 1, 2, \ldots$ and $0 < P < 1$.

The geometric distribution as defined above has mean 1/P.

The i-th geometric deviate is generated as the smallest integer not less than $(\log (U_i))/(\log (1 - P))$, where the U_i are independent uniform(0, 1) random numbers (see Knuth 1981).

The geometric distribution is often defined on 0, 1, 2, ..., with mean $(1 - P)/P$. Such deviates can be obtained by subtracting 1 from each element of the returned vector of random deviates.

Hypergeometric Distribution

Calling RANDOM with keywords *Hypergeometric,* and *Parameter=*[*M, N, L,*] generates pseudorandom numbers from a hypergeometric distribution with parameters *N*, *M*, and *L*. The hypergeometric random variable *X* can be thought of as the number of items of a given type in a random sample of size *N* that is drawn without replacement from a population of size *L* containing *M* items of this type. The probability function is

$$f(x) = \frac{\binom{M}{x}\binom{L-M}{N-x}}{\binom{L}{N}}$$

for $x = \max(0, N - L + M), 1, 2, \ldots, \min(N, M)$

If the hypergeometric probability function with parameters *N*, *M*, and *L* evaluated at $N - L + M$ (or at 0 if this is negative) is greater than the machine, and

less than 1.0 minus the machine epsilon, then RANDOM uses the inverse CDF technique. The routine recursively computes the hypergeometric probabilities, starting at $x = \max(0, N - L + M)$ and using the ratio

$$\frac{f(X = x+1)}{f(X = x)}$$

(see Fishman 1978, p. 475).

If the hypergeometric probability function is too small or too close to 1.0, then RANDOM generates integer deviates uniformly in the interval $[1, L - i]$ for $i = 0, 1, ...$, and at the i-th step, if the generated deviate is less than or equal to the number of special items remaining in the lot, the occurrence of one special item is tallied and the number of remaining special items is decreased by one. This process continues until the sample size of the number of special items in the lot is reached, whichever comes first. This method can be much slower than the inverse CDF technique. The timing depends on N. If N is more than half of L (which in practical examples is rarely the case), the user may wish to modify the problem, replacing N by $L - N$, and to consider the generated deviates to be the number of special items not included in the sample.

Logarithmic Distribution

Calling RANDOM with keywords *Logarithmic* and *Parameter=a* generates pseudorandom numbers from a logarithmic distribution. The probability function is

$$f(x) = -\frac{a^x}{x \ln(1-a)}$$

for $x = 1, 2, 3, ...$, and $0 < a < 1$

The methods used are described by Kemp (1981) and depend on the value of a. If a is less than 0.95, Kemp's algorithm LS, which is a "chop-down" variant of an inverse CDF technique, is used. Otherwise, Kemp's algorithm LK, which gives special treatment to the highly probable values of 1 and 2 is used.

Lognormal Distribution

Calling RANDOM with keywords *Lognormal,* and *Parameter*=[μ, σ] generates pseudorandom numbers from a lognormal distribution. The scale parameter σ in the underlying normal distribution must be positive. The method is to generate

normal deviates with mean μ and standard deviation Σ and then to exponentiate the normal deviates.

The probability density function for the lognormal distribution is

$$f(x) = \frac{1}{\sigma x\sqrt{2\pi}} \exp\left[-\frac{1}{2\sigma^2}(\ln x - \mu)^2\right]$$

for $x > 0$. The mean and variance of the lognormal distribution are $\exp(\mu + \sigma^2/2)$ and $\exp(2\mu + 2\sigma^2) - \exp(2\mu + \sigma^2)$, respectively.

Negative Binomial

Calling RANDOM with keywords *Neg_binomial* and *Parameters*=[*r, p*] generates pseudorandom numbers from a negative binomial distribution. The parameters *r* and *p* must be positive and *p* must be less than 1. The probability function is

$$f(x) = \binom{r+x-1}{x}(1-p)^r p^x$$

for $x = 0, 1, 2, ...$

If *r* is an integer, the distribution is often called the Pascal distribution and can be thought of as modeling the length of a sequence of Bernoulli trials until *r* successes are obtained, where *p* is the probability of getting a success on any trial. In this form, the random variable takes values *r*, *r* + 1, *r* + 2, … and can be obtained from the negative binomial random variable defined above by adding *r* to the negative binomial variable defined by adding *r* to the negative binomial variable. This latter form is also equivalent to the sum of *r* geometric random variables defined as taking values 1, 2, 3, ...

If $rp/(1-p)$ is less than 100 and $(1-p)^r$ is greater than the machine epsilon, RANDOM uses the inverse CDF technique; otherwise, for each negative binomial deviate, RANDOM generates a *gamma* $(r, p/(1-p))$ deviate *Y* and then generates a Poisson deviate with parameter *Y.*

Discrete Uniform Distribution

Calling RANDOM with keywords *Discrete_unif* and *Parameters=k* generates pseudorandom numbers from a uniform discrete distribution over the integers 1, 2, ..., *k*. A random integer is generated by multiplying *k* by a uniform (0, 1) ran-

dom number, adding 1.0, and truncating the result to an integer. This, of course, is equivalent to sampling with replacement from a finite population of size *k*.

Student's t Distribution

Calling RANDOM with keywords *Students_t* and *Parameters=Df* generates pseudorandom numbers from a Student's *t* distribution with *Df* degrees of freedom, using a method suggested by Kinderman et al. (1977). The method ("TMX" in the reference) involves a representation of the *t* density as the sum of a triangular density over (−2, 2) and the difference of this and the *t* density. The mixing probabilities depend on the degrees of freedom of the *t* distribution. If the triangular density is chosen, the variate is generated as the sum of two uniforms; otherwise, an acceptance/rejection method is used to generate the difference density.

Triangular Distribution

Calling RANDOM with the keyword *Triangular* generates pseudorandom numbers from a triangular distribution over the unit interval. The probability density function is $f(x) = 4x$, for $0 \le x \le 0.5$, and $f(x) = 4(1 - x)$, for $0.5 < x \le 1$. An inverse CDF technique is used.

von Mises Distribution

Calling RANDOM with keywords *Von_mises* and *Parameters=c* generates pseudorandom numbers from a von Mises distribution where *c* must be positive. The probability density function is

$$f(x) = \frac{1}{2\pi I_0(c)} \exp\left[c\cos(x)\right]$$

for $-\pi < x < \pi$, where $I_0(c)$ is the modified Bessel function of the first kind of order 0. The probability density is equal to 0 outside the interval $(-\pi, \pi)$.

The algorithm is an acceptance/rejection method using a wrapped Cauchy distribution as the majorizing distribution. It is due to Nest and Fisher (1979).

Weibull Distribution

Calling RANDOM with keywords *Weibull* and *Parameters*=[*a*,*b*] generates pseudorandom numbers from a Weibull distribution with shape parameter *a* and scale parameter *b*. The probability density function is

$$f(x) = abx^{a-1}\exp\left(-bx^{a}\right)$$

for x^3 0, $a > 0$, and $b > 0$. The value of b is optional; if it is not specified, it is set to 1.0.

Function RANDOM uses an antithetic inverse CDF technique to generate a Weibull variate; that is, a uniform random deviate U is generated and the inverse of the Weibull cumulative distribution function is evaluated at $1.0 - U$ to yield the Weibull deviate.

Note that the Rayleigh distribution with probability density function

$$r(x) = \frac{1}{\alpha^2} xe^{-\left(x^2/\left(2\alpha^2\right)\right)}$$

for $x \geq 0$ is the same as a Weibull distribution with shape parameter a equal to 2 and scale parameter b equal to

$$\sqrt{2}\alpha$$

Stable Distribution

Calling RANDOM with keywords Stable and *Parameters*=[α, β'] generates pseudorandom numbers from a stable distribution with parameters α‘ and β'. α is the usual characteristic exponent parameter α and β' is related to the usual skewness parameter β of the stable distribution. With the restrictions $0 < \alpha \leq 2$ and $-1 \leq \beta \leq 1$, the characteristic function of the distribution is

j(t) = exp[-| t |a exp(-pib(1 - |1 - a|)sign(t)/2)] for a ¼ 1

and

j(t) = exp[-| t |(1 + 2ib ln| t |)sign(t)/p)] for a = 1

When $\beta = 0$, the distribution is symmetric. In this case, if $\alpha = 2$, the distribution is normal with mean 0 and variance 2; and if $\alpha = 1$, the distribution is Cauchy.

The parameterization using β' and the algorithm used here are due to Chambers, Mallows, and Stuck (1976). The relationship between β' and the standard β is

$\beta' = -\tan(\pi(1 - \alpha)/2)\ \tan(-\pi\beta(1 - |1 - \alpha|)/2)$ for $\alpha \neq 1$

and

$\beta' = \beta$ for $\alpha = 1$

The algorithm involves formation of the ratio of a uniform and an exponential random variate.

Multinomial Distribution

Calling RANDOM with keywords *Multinomial, Probabilites,* and *Parameters=ntrials* generates pseudorandom numbers from a *K*-variate multinomial distribution with parameters *n* and *p*. *k*=N_ELEMENTS(*Probabilities*) and *ntrials* must be positive. Each element of *Probabilites* must be positive and the elements must sum to 1. The probability function
(with $n = n$, $k = k$, and $p_i = Probabilities(i)$) is

$$f(x_1, x_2, \ldots, x_k) = \frac{n!}{x_1! x_2! \ldots x_k!} p_1^{x_1} p_2^{x_2} \ldots p_k^{x_k}$$

for $x_i \geq 0$ and

$$\sum_{i=0}^{k-1} x_i = n$$

The deviate in each row of *r* is produced by generation of the binomial deviate x_0 with parameters *n* and p_i and then by successive generations of the conditional binomial deviates x_j given $x_0, x_1, \ldots, x_{j-2}$ with parameters $n - x_0 - x_1 - \ldots - x_{j-2}$ and $p_j / (1 - p_0 - p_1 - \ldots - p_{j-2})$.

Random Points on a K-dimensional Sphere

Calling RANDOM with the keywords *Sphere* and *Parameters= k* generates pseudorandom coordinates of points that lie on a unit circle or a unit sphere in *K*-dimensional space. For points on a circle ($k = 2$), pairs of uniform (– 1, 1) points are generated and accepted only if they fall within the unit circle (the sum of their squares is less than 1), in which case they are scaled so as to lie on the circle.

For spheres in three or four dimensions, the algorithms of Marsaglia (1972) are used. For three dimensions, two independent uniform (– 1, 1) deviates U_1 and

U_2 are generated and accepted only if the sum of their squares S_1 is less than 1. Then, the coordinates

$$Z_1 = 2U_1\sqrt{1-S_1},\ Z_2 = 2U_2\sqrt{1-S_1},\ \text{and}\ Z_3 = 1-2S_1$$

are formed. For four dimensions, U_1, U_2, and S_1 are produced as described above. Similarly, U_3, U_4, and S_2 are formed. The coordinates are then

$$Z_1 = U_1,\ Z_2 = U_2,\ Z_3 = U_3\sqrt{(1-S_1)/S_2}$$

and

$$Z_4 = U_4\sqrt{(1-S_1)/S_2}$$

For spheres in higher dimensions, *K* independent normal deviates are generated and scaled so as to lie on the unit sphere in the manner suggested by Muller (1959).

Random Permutation

Calling RANDOM with the keyword *Permutation* generates a pseudorandom permutation of the integers from 1 to *n*. It begins by filling a vector of length *n* with the consecutive integers 1 to *n*. Then, with *M* initially equal to *n*, a random index *J* between 1 and *M* (inclusive) is generated. The element of the vector with the index *M* and the element with index *J* swap places in the vector. *M* is then decremented by 1 and the process repeated until *M* = 1.

Sample Indices

Calling RANDOM with the keywords *Sample_indices* and *Parameters=npop* generates the indices of a pseudorandom sample,without replacement, of size *n* numbers from a population of size *npop*. If *n* is greater than *npop*/2, the integers from 1 to *npop* are selected sequentially with a probability conditional on the number selected and the number remaining to be considered. If, when the *i*-th population index is considered, *j* items have been included in the sample, then the index *i* is included with probability $(n - j)/(npop + 1 - i)$.

If *n* is not greater than *npop*/2, a $O(n)$ algorithm due to Ahrens and Dieter (1985) is used. Of the methods discussed by Ahrens and Dieter, the one called SG⋆ is used. It involves a preliminary selection of *q* indices using a geometric distribution for the distances between each index and the next one. If the pre-

liminary sample size q is less than n, a new preliminary sample is chosen, and this is continued until a preliminary sample greater in size than n is chosen. This preliminary sample is then thinned using the same kind of sampling as described above for the case in which the sample size is greater than half of the population size. This routine does not store the preliminary sample indices, but rather restores the state of the generator used in selecting the sample initially, and then passes through once again, making the final selection as the preliminary sample indices are being generated.

Example 1

In this example, RANDOM is used to generate five pseudorandom, uniform numbers. Since RANDOMOPT is not called, the generator used is a simple multiplicative congruential one with a multiplier of 16807.

```
RANDOMOPT, Set = 123457
    ; Set the random seed.
r   = RANDOM(5)
    ; Call RANDOM to compute the random numbers.
PM, r
    ; Output the results.
0.966220
0.260711
0.766262
0.569337
0.844829
```

Example 2: Poisson Distribution

In this example, random numbers from a Poisson distribution are computed.

```
RANDOMOPT, Set = 123457
r = RANDOM(5, /Poisson, Parameters = 0.5)
    ; Call RANDOM with keywords Poisson and Theta.
PM, r
 2
 0
 1
 0
 1
```

Example 3: Beta Distribution

In this example, random numbers are computed from a Beta distribution.

```
RANDOMOPT, set = 123457

r = RANDOM(5, /Beta, Parameter = [3,2])
    ; Call RANDOM with keywords Beta, Pin, and Qin.

PM, r

 0.281392
 0.948276
 0.398391
 0.310306
 0.829578
```

Example 4: Scaling the Results of RANDOM

This example computes deviates with uniform density over the interval (10, 20) and deviates from the normal distribution with a mean of 10 and a standard deviation of 2.

```
RANDOMOPT, Set = 123457
    ; Set the random number seed.

a = 10
    ; Define the lowerbound.

b = 20
    ; Define the upperbound.

r   = a + (b - a) * RANDOM(5)
    ; Call RANDOM to compute the deviates on (0,1) and scale the
    ; results to (a,b).

PM, r
    ; Output the results.

19.6622
   12.6071
   17.6626
   15.6934
   18.4483

stdev = 2
    ; Define a standard deviation.

mean =  10
    ; Define a mean.

r = RANDOM(6, /Normal) * stdev + mean
```

```
    ; Call RANDOM to compute the deviates normal deviates and scale
    ; the results using the specified mean and standard deviation.
PM, r
    ; Output the results.
   6.59363
   14.4635
   10.5137
   12.5223
   9.39352
   5.71021
```

Example 5: Multivariate Normal Distribution

In this example, RANDOM generates five pseudorandom normal vectors of length 2 with variance covariance matrix equal to the following:

$$\begin{bmatrix} 0.500 & 0.375 \\ 0.375 & 0.500 \end{bmatrix}$$

```
RANDOMOPT, Set = 123457
    ; Set the random number seed.
RM, cov, 2, 2
    ; Read the covariance matrix.
row 0: .5    .375
row 1: .375 .5
PM, RANDOM(5, /Mvar_Normal, Covariances = cov)
  1.45068        1.24634
  0.765975     -0.0429410
  0.0583781    -0.669214
  0.903489       0.462826
 -0.866886      -0.933426
```

FAURE_INIT Function

Initializes the structure used for computing a shuffled Faure sequence.

Usage

result = FAURE_INIT(*ndim*)

Input Parameters

ndim — The dimension of the hyper-rectangle.

Returned Value

A structure that contains information about the sequence.

Input Keywords

Base — The base of the Faure sequence.
Default: The smallest prime greater than or equal to *ndim.*

Skip — The number of points to be skipped at the beginning of the Faure sequence. Default:

$$\left\lfloor base^{m/2-1} \right\rfloor$$

where

$$m = \left\lfloor \log\ \mathrm{B} / \log\ base \right\rfloor$$

and *B* is the largest representable integer.

Discussion

Discrepancy measures the deviation from uniformity of a point set.

The discrepancy of the point set

$$x_1, \ldots, x_n \in [0,1]^d, d \geq 1,$$

is

$$D_n^{(d)} = \sup_E \left| \frac{A(E;n)}{n} - \lambda(E) \right|,$$

where the supremum is over all subsets of [0, 1]^d of the form

$$E = \left[0, t_1\right) \times \ldots \times \left[0, t_d\right),\ 0 \leq t_j \leq 1,\ 1 \leq j \leq d,$$

λ is the Lebesque measure, and

$$(E; n$$

is the number of the x_j contained in E.

The sequence $x_1, x_2, \ldots$ of points $[0,1]^d$ is a low-discrepancy sequence if there exists a constant $c(d)$, depending only on d, such that

$$D_n^{(d)} \le c(d)\frac{(\log n)^d}{n}$$

for all $n>1$.

Generalized Faure sequences can be defined for any prime base $b \ge d$. The lowest bound for the discrepancy is obtained for the smallest prime $b \ge d$, so the keyword *Base* defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence $x_1, x_2, \ldots$, is computed as follows:

Write the positive integer n in its *b-ary* expansion,

$$n = \sum_{i=0}^{\infty} a_i(n) b^i$$

where $a_i(n)$ are integers,

$$0 \le a_i(n) < b$$

The *j-th* coordinate of x_n is

$$x_n^{(j)} = \sum_{k=0}^{\infty}\sum_{d=0}^{\infty} c_{kd}^{(j)}\ a_d(n)\ b^{-k-1}, \qquad 1 \le j \le d$$

The generator matrix for the series,

$$c_{k\,d}^{(j)}$$

is defined to be

$$c_{k\,d}^{(j)} = j^{\,d-k} c_{k\,d}$$

and

$$c_{k\,d}$$

is an element of the Pascal matrix,

$$c_{k\,d} = \begin{cases} \dfrac{d!}{c!(d-c)!} & k \le d \\ 0 & k > d \end{cases}$$

It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the *b-ary* Gray code. The function *G(n)* maps the positive integer *n* into the integer given by its *b-ary* expansion.

The sequence computed by this function is *x*(*G*(*n*)), where *x* is the generalized Faure sequence.

Example

In this example, five points in the Faure sequence are computed. The points are in the three-dimensional unit cube.

Note that FAURE_INIT is used to create a structure that holds the state of the sequence. Each call to FAURE_NEXT_PT returns the next point in the sequence and updates the state structure.

```
state = FAURE_INIT(3)
p = FAURE_NEXT_PT(5, state)
PM, p
      0.333689      0.492659     0.0640654
      0.667022      0.825992      0.397399
      0.778133      0.270436      0.175177
      0.111467      0.603770      0.508510
```

```
    0.444800       0.937103       0.841843
```

FAURE_NEXT_PT Function

Computes a shuffled Faure sequence.

Usage

result = FAURE_NEXT_PT(*npts*, *state*)

Input Parameters

npts — The number of points to generate in the hyper-rectangle.

state — State structure created by a call to FAURE_INIT.

Returned Value

An array of size *npts* by *state.dim* containing the *npts* next points in the shuffled Faure sequence.

Input Keywords

Double — If present and nonzero, double precision is used.

Output Keywords

Skip — The current point in the sequence. The sequence can be restarted by initializing a new sequence using this value for *Skip*, and using the same dimension for *ndim*.

Discussion

Discrepancy measures the deviation from uniformity of a point set.

The discrepancy of the point set

$$x_1, \ldots, x_n \in [0,1]^d, d \geq 1,$$

is

$$D_n^{(d)} = \sup_E \left| \frac{A(E;n)}{n} - \lambda(E) \right|,$$

where the supremum is over all subsets of [0, 1]^d of the form

$$E = [0, t_1) \times \ldots \times [0, t_d), \ 0 \leq t_j \leq 1, \ 1 \leq j \leq d,$$

λ is the Lebesque measure, and

$$(E; n$$

is the number of the x_j contained in *E*.

The sequence $x_1, x_2, \ldots$ of points [0,1]^d is a low-discrepancy sequence if there exists *a* constant *c(d)*, depending only on *d*, such that

$$D_n^{(d)} \leq c(d) \frac{(\log n)^d}{n}$$

for all $n>1$.

Generalized Faure sequences can be defined for any prime base $b \geq d$. The lowest bound for the discrepancy is obtained for the smallest prime $b \geq d$, so the keyword Base defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence $x_1, x_2, \ldots$, is computed as follows:

Write the positive integer *n* in its *b-ary* expansion

$$n = \sum_{i=0}^{\infty} a_i(n) b^i$$

where $a_i(n)$ are integers,

$$0 \le a_i(n) < b$$

The j-th coordinate of x_n is

$$x_n^{(j)} = \sum_{k=0}^{\infty} \sum_{d=0}^{\infty} c_{kd}^{(j)} \; a_d(n) \; b^{-k-1}, \qquad 1 \le j \le d$$

The generator matrix for the series,

$$c_{k\,d}^{(j)},$$

is defined to be

$$c_{k\,d}^{(j)} = j^{\,d-k} c_{k\,d}$$

and

$$c_{k\,d}$$

is an element of the Pascal matrix,

$$c_{k\,d} = \begin{cases} \dfrac{d!}{c!(d-c)!} & k \le d \\ 0 & k > d \end{cases}$$

It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the *b-ary* Gray cod*e*. The function *G(n*) maps the positive integer *n in*to the integer given by its *b-ary* expansion.

The sequence computed by this function is $x(G(n))$, where x is the generalized Faure sequence.

Example

In this example, five points in the Faure sequence are computed. The points are in the three-dimensional unit cube.

Note that FAURE_INIT is used to create a structure that holds the state of the sequence. Each call to FAURE_NEXT_PT returns the next point in the sequence and updates the state structure.

```
state = FAURE_INIT(3)

p = FAURE_NEXT_PT(5, state)

PM, p
      0.333689      0.492659     0.0640654
      0.667022      0.825992      0.397399
      0.778133      0.270436      0.175177
      0.111467      0.603770      0.508510
      0.444800      0.937103      0.841843
```

11

CHAPTER

Probability Distribution Functions and Inverses

Contents of Chapter

NORMALCDF Function

Evaluates the standard normal (Gaussian) distribution function. Using a keyword, the inverse of the standard normal (Gaussian) distribution can be evaluated.

Usage

result = NORMALCDF(*x*)

Input Parameters

x — Expression for which the normal distribution function is to be evaluated.

Returned Value

result — The probability that a normal random variable takes a value less than or equal to *x*.

Input Keywords

Double — If present and nonzero, double precision is used.

Inverse — If present and nonzero, evaluates the inverse of the standard normal (Gaussian) distribution function. If *Inverse* is specified, then argument *x* represents the probability for which the inverse of the normal distribution function is to be evaluated. In this case, *x* must be in the open interval (0.0, 1.0).

Discussion

Function NORMALCDF evaluates the distribution function, Φ, of a standard normal (Gaussian) random variable; that is,

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

The value of the distribution function at the point *x* is the probability that the random variable takes a value less than or equal to *x*.

The standard normal distribution (for which NORMALCDF is the distribution function) has mean of zero and variance of 1. The probability that a normal ran-

dom variable with mean μ and variance σ^2 is less than y is given by NORMALCDF evaluated at $(y - \mu) / \sigma$.

The function $\Phi(x)$ is evaluated by use of the complementary error function, ERFC (page 358). The relationship follows below.

$$\Phi(x) = \text{ERFC}((-x/\sqrt{2.0})/2.)$$

If the keyword *Inverse* is specified, the NORMALCDF function evaluates the inverse of the distribution function, Φ, of a standard normal (Gaussian) random variable; that is,

NORMALCDF (x, */Inverse*) = $\Phi^{-1}(x)$ where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-t^2/2}dt$$

The value of the distribution function at the point x is the probability that the random variable takes a value less than or equal to x. The standard normal distribution has a mean of zero and a variance of 1.

The NORMALCDF function is evaluated by use of minimax rational-function approximations for the inverse of the error function. General descriptions of these approximations are given in Hart et al. (1968) and Strecok (1968). The rational functions used in NORMALCDF are described by Kinnucan and Kuki (1968).

Example

Suppose X is a normal random variable with mean 100 and variance 225. This example finds the probability that X is less than 90 and the probability that X is between 105 and 110.

```
x1 = (90-100)/15.
p = NORMALCDF(x1)
PM, p, Title = $
   'The probability that X is less than 90 is:'
The probability that X is less than 90
   is: 0.252493
x1 = (105 - 100)/15.
x2 = (110 - 100)/15.
p = NORMALCDF(x2) - NORMALCDF(x1)
PM, p, Title = $
```

```
'The probability that X is between 105 and ', $
   '110 is:'
The probability that X is between 105 and 110
   is: 0.116949
```

BINORMALCDF Function

Evaluates the bivariate normal distribution function.

Usage

result = BINORMALCDF(*x*, *y*, *rho*)

Input Parameters

x — The *x*-coordinate of the point for which the bivariate normal distribution function is to be evaluated.

y — The *y*-coordinate of the point for which the bivariate normal distribution function is to be evaluated.

rho — Correlation coefficient.

Returned Value

result — The probability that a bivariate normal random variable with correlation *rho* takes a value less than or equal to *x* and less than or equal to *y*.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function BINORMALCDF evaluates the distribution function *F* of a bivariate normal distribution with means of zero, variances of 1, and correlation of *rho*; that is, ρ = *rho* and $|\rho| < 1$.

$$F(x,y) = \frac{1}{2\pi\sqrt{1-\rho^2}}\int_{-\infty}^{x}\int_{-\infty}^{y}\exp\left(-\frac{u^2-2\rho uv+v^2}{2(1-\rho^2)}\right)dudv$$

To determine the probability that $U \le u_0$ and $V \le v_0$, where (U, V) is a bivariate normal random variable with mean $\mu = (\mu_U, \mu_V)$ and the following variance-covariance matrix:

$$\sum = \begin{bmatrix} \sigma_U^2 & \sigma_{UV} \\ \sigma_{UV} & \sigma_V^2 \end{bmatrix}$$

transform $(U, V)^T$ to a vector with zero means and unit variances. The input to BINORMALCDF would be as follows:

$$X = \frac{(u_0 - \mu_U)}{\sigma_U}, \quad Y = \frac{(v_0 - \mu_V)}{\sigma_V}, \text{ and } \rho = \frac{\sigma_{UV}}{(\sigma_U \sigma_V)}$$

The BINORMALCDF function uses the method of Owen (1962, 1965). For $|\rho| = 1$, the distribution function is computed based on the univariate statistic $Z = \min(x, y)$ and on the normal distribution NORMALCDF.

Example

Suppose (x, y) is a bivariate normal random variable with mean (0, 0) and the following variance-covariance matrix:

$$\begin{bmatrix} 1.0 & 0.9 \\ 0.9 & 1.0 \end{bmatrix}$$

This example finds the probability that x is less than –2.0 and y is less than 0.0.

```
x = -2
y = 0
rho = .9
    ; Define x, y, and rho.
p = BINORMALCDF(x, y, rho)
    ; Call BINORMALCDF and output the results.
PM, 'P((x < -2.0) and (y < 0.0)) = ', p, $
   Format = '(a29, f8.4)'
P((x < -2.0) and (y < 0.0)) = 0.0228
```

CHISQCDF Function

Evaluates the chi-squared distribution or noncentral chi-squared distribution. Using a keyword the inverse of these distributions can be computed.

Usage

result = CHISQCDF(*chisq, df [, delta]*)

Input Parameters

chisq — Expression for which the chi-squared distribution function is to be evaluated.

df — Number of degrees of freedom of the chi-squared distribution. Argument *df* must be greater than or equal to 0.5.

delta — (Optional) The noncentrality parameter. *delta* must be nonnegative, and *delta* + *df* must be less than or equal to 200,000.

Returned Value

result — The probability that a chi-squared random variable takes a value less than or equal to *chisq*.

Input Keywords

Double — If present and nonzero, double precision is used.

Inverse — If present and nonzero, evaluates the inverse of the chi-squared distribution function. If inverse is specified, then argument *chisq* represents the probability for which the inverse of the chi-squared distribution function is to be evaluated. Argument *chisq* must be in the open interval (0.0, 1.0).

Discussion

If Two Input Arguments Are Used

Function CHISQCDF evaluates the distribution function, *F*, of a chi-squared random variable with $\nu = df$. Then,

$$F(x) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)}\int_0^x e^{-t/2}t^{\nu/2-1}dt$$

where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function at the point *x* is the probability that the random variable takes a value less than or equal to *x*.

For $\nu > 65$, CHISQCDF uses the Wilson-Hilferty approximation (Abramowitz and Stegun 1964, Equation 26.4.17) to the normal distribution, and NORMALCDF function is used to evaluate the normal distribution function.

For $\nu \leq 65$, CHISQCDF uses series expansions to evaluate the distribution function. If $x < \max(\nu / 2, 26)$, CHISQCDF uses the series 6.5.29 in Abramowitz and Stegun (1964); otherwise, it uses the asymptotic expansion 6.5.32 in Abramowitz and Stegun.

If *Inverse* is specified, the CHISQCDF function evaluates the inverse distribution function of a chi-squared random variable with $\nu = df$ and with probability *p*. That is, it determines *x*, such that

$$p = \frac{1}{2^{\nu/2}\Gamma(\nu/2)}\int_0^x e^{-t/2}t^{\nu/2-1}dt$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to *x* is *p*.

For $\nu < 40$, CHISQCDF uses bisection (if $\nu \leq 2$ or $p > 0.98$) or regula falsi to find the expression for which the chi-squared distribution function is equal to *p*.

For $40 \leq \nu < 100$, a modified Wilson-Hilferty approximation (Abramowitz and Stegun 1964, Equation 26.4.18) to the normal distribution is used. The NORMALCDF function is used to evaluate the inverse of the normal distribution function. For $\nu \geq 100$, the ordinary Wilson-Hilferty approximation (Abramowitz and Stegun 1964, Equation 26.4.17) is used.

If Three Input Arguments Are Used

Function CHISQCDF evaluates the distribution function of a noncentral chi-squared random variable with `df` degrees of freedom and noncentrality parameter `alam`, that is, with ν = `df`, λ = `alam`, and x = `chi_squared`,

$$non\_central\_chi\_sq(x) = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2}(\lambda/2)^i}{i!} \int_0^x \frac{t^{(\nu+2i)/2-1} e^{-t/2}}{2^{\nu+2i)/2\Gamma(\frac{\nu+2i}{2})}} dt$$

where $\Gamma(\cdot)$ is the gamma function. This is a series of central chi-squared distribution functions with Poisson weights. The value of the distribution function at the point x is the probability that the random variable takes a value less than or equal to x.

The noncentral chi-squared random variable can be defined by the distribution function above, or alternatively and equivalently, as the sum of squares of independent normal random variables. If *Yi* have independent normal distributions with means μi and variances equal to one and

$$X = \sum_{i=1}^{n} Y_i^2$$

then X has a noncentral chi-squared distribution with n degrees of freedom and noncentrality parameter equal to

$$\sum_{i=1}^{n} \mu_i^2$$

With a noncentrality parameter of zero, the noncentral chi-squared distribution is the same as the chi-squared distribution.

Function CHISQCDF determines the point at which the Poisson weight is greatest, and then sums forward and backward from that point, terminating when the additional terms are sufficiently small or when a maximum of 1000 terms have been accumulated. The recurrence relation 26.4.8 of Abramowitz and Stegun (1964) is used to speed the evaluation of the central chi-squared distribution functions.

If *Inverse* is specified, CHISQCDF evaluates the inverse distribution function of a noncentral chi-squared random variable with `df` degrees of freedom and noncentrality parameter `delta`; that is, with P = `p`, ν = `df`, and λ = `delta`, it determines c_0 (= CHISQCDF(`p, df, delta`)), such that

$$P = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2}(\lambda/2)^i}{i!} \int_0^{c_0} \frac{x^{(\nu+2i)/2-1} e^{-x/2}}{2^{(\nu+2i)/2}\,\Gamma(\frac{\nu+2i}{2})} dx$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to c_0 is *P*.

Example

Suppose *X* is a chi-squared random variable with two degrees of freedom. This example finds the probability that *X* is less than 0.15 and the probability that *X* is greater than 3.0.

```
df = 2
chisq = .15
p = CHISQCDF(chisq, df)

PM, p, Title = $
   'The probability that chi-squared ' + $
   'with 2 df is less than .15 is:'

The probability that chi-squared with 2 df is
   less than .15 is: 0.0722565

df = 2
chisq = 3
p = 1 - CHISQCDF(chisq, df)

PM, p, Title = $
   'The probability that chi-squared ' + $
   'with 2 df is greater than 3 is:'

The probability that chi-squared with 2 df
   is greater than 3 is: 0.223130
```

Informational Errors

STAT_ARG_LESS_THAN_ZERO — Input parameter, *chisq*, is less than zero.

STAT_UNABLE_TO_BRACKET_VALUE — Bounds that enclose *p* could not be found. An approximation for CHISQCDF is returned.

STAT_CHI_2_INV_CDF_CONVERGENCE — Value of the inverse chi-squared could not be found within a specified number of iterations. An approximation for CHISQCDF is returned.

Alert Errors

STAT_NORMAL_UNDERFLOW — Using the normal distribution for large degrees of freedom, underflow would have occurred.

FCDF Function

Evaluates the *F* distribution function. Using a keyword, the inverse of the *F* distribution function can be evaluated.

Usage

result = FCDF(*f*, *dfnum*, *dfden*)

Input Parameters

f — Expression for which the *F* distribution function is to be evaluated.

dfnum — Numerator degrees of freedom. Argument *dfnum* must be positive.

dfden — Denominator degrees of freedom. Argument *dfden* must be positive.

Returned Value

result — The probability that an *F* random variable takes a value less than or equal to the input point *f*.

Input Keywords

Double — If present and nonzero, double precision is used.

Inverse — If present and nonzero, evaluates the inverse of the *F* distribution function. If inverse is specified, argument *f* represents the probability for which the inverse of the *F* distribution function is to be evaluated. In this case, *f* must be in the open interval (0.0, 1.0).

Discussion

Function FCDF evaluates the distribution function of a Snedecor's *F* random variable with *dfnum* and *dfden*. The function is evaluated by making a transformation to a beta random variable and then evaluating the incomplete beta function. If *X* is an *F* variate with ν_1 and ν_2 degrees of freedom and

$Y = (\nu_1 X) / (\nu_2 + \nu_1 X)$, then Y is a beta variate with parameters $p = \nu_1 / 2$ and $q = \nu_2 / 2$. The FCDF function also uses a relationship between F random variables that can be expressed as follows:
$F_F(f, \nu_1, \nu_2) = 1 - F_F(1 / f, \nu_2, \nu_1)$, where F_F is the distribution function for an F random variable.

If the keyword *Inverse* is specified, the FCDF function evaluates the inverse distribution function of a Snedecor's F random variable with $\nu_1 =$ *dfnum* numerator degrees of freedom and $\nu_2 =$ *dfden* denominator degrees of freedom. The function is evaluated by making a transformation to a beta random variable and then evaluating the inverse of an incomplete beta function.

Example

This example finds the probability that an F random variable with one numerator and one denominator degree of freedom is greater than 648.

```
f = 648
p = 1 - FCDF(f, 1, 1)
PM, p, Title = $
   'The probability that an F(1,1) ' + $
   'variate is greater than 648 is:'
The probability that an F(1,1) variate is greater than 648 is:
   0.0249959
```

Fatal Errors

STAT_F_INVERSE_OVERFLOW — Function FCDF is set to machine infinity since overflow would occur upon modifying the inverse value for the F distribution with the result obtained from the inverse beta distribution.

TCDF Function

Evaluates the Student's *t* distribution or noncentral Student's *t* distribution. Using a keyword the inverse of these distributions can be computed.

Usage

result = TCDF(*chisq, df [, delta]*)

Input Parameters

t — Argument for which the Student's *t* distribution function is to be evaluated.

df — Degrees of freedom. Argument *df* must be greater than or equal to 1.0.

delta — (Optional) The noncentrality parameter.

Returned Value

result — The probability that a Student's *t* random variable takes a value less than or equal to the input *t*.

Input Keywords

Double — If present and nonzero, double precision is used.

Inverse — If present and nonzero, evaluates the inverse of the Student's *t* distribution function. If *Inverse* is specified, argument *t* represents the probability for which the inverse of the Student's *t* distribution function is to be evaluated. In this case, *t* must be in the open interval (0.0, 1.0).

Discussion

If Two Input Arguments Are Used

Function TCDF evaluates the distribution function of a Student's *t* random variable with $\nu = df$ degrees of freedom. If $t^2 \geq \nu$, the relationship of a *t* to an *F* random variable (and subsequently, to a beta random variable) is exploited, and percentage points from a beta distribution are used. Otherwise, the method described by Hill (1970) is used. If ν is not an integer or if ν is greater than 19, a Cornish-Fisher expansion is used to evaluate the distribution function. If ν is less than 20 and $|t|$ is less than 2.0, a trigonometric series (see Abramowitz and

Stegun 1964, Equations 26.7.3 and 26.7.4, with some rearrangement) is used. For the remaining cases, a series given by Hill (1970) that converges well for large values of *t* is used.

If keyword *Inverse* is specified, the TCDF function evaluates the inverse distribution function of a Student's *t* random variable with $\nu = df$ degrees of freedom. If ν equals 1 or 2, the inverse can be obtained in closed form. If ν is between 1 and 2, the relationship of a *t* to a beta random variable is exploited, and the inverse of the beta distribution is used to evaluate the inverse. Otherwise, the algorithm of Hill (1970) is used. For small values of ν greater than 2, Hill's algorithm inverts an integrated expansion in $1 / (1 + t^2 / \nu)$ of the *t* density. For larger values, an asymptotic inverse Cornish-Fisher type expansion about normal deviates is used.

If Three Input Arguments Are Used

Function TCDF evaluates the distribution function *F* of a noncentral *t* random variable with `df` degrees of freedom and noncentrality parameter `delta`; that is, with v = `df`, δ = `delta` , and t_0 = `t`,

$$F(t_0) = \int_{-\infty}^{t_0} \frac{v^{v/2} e^{-\delta^2/2}}{\sqrt{\pi}\Gamma(v/2)(v+x^2)^{(v+1)/2}} \sum_{i=0}^{\infty} \Gamma((v+i+1)/2)\left(\frac{\delta^i}{i!}\right)\left(\frac{2x^2}{v+x^2}\right)^{i/2} dx$$

where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function at the point t_0 is the probability that the random variable takes a value less than or equal to t_0.

The noncentral *t* random variable can be defined by the distribution function above, or alternatively and equivalently, as the ratio of a normal random variable and an independent chi-squared random variable. If *w* has a normal distribution with mean δ and variance equal to one, *u* has an independent chi-squared distribution with *v* degrees of freedom, and

$$x = w / \sqrt{u / v}$$

then *x* has a noncentral *t* distribution with degrees of freedom and noncentrality parameter δ.

The distribution function of the noncentral *t* can also be expressed as a double integral involving a normal density function (see, for example, Owen 1962, page 108). The function `TNDF` uses the method of Owen (1962, 1965), which

uses repeated integration by parts on that alternate expression for the distribution function.

If Inverse is specified TCDF evaluates the inverse distribution function of a noncentral *t* random variable with `df` degrees of freedom and noncentrality parameter `delta`; that is, with P = `p`, v = `df`, and δ = `delta`, it determines t_0 (= TCDF(`p, df, delta`)), such that

$$P = \int_{-\infty}^{t_0} \frac{v^{v/2} e^{-\delta^2/2}}{\sqrt{\pi}\Gamma(v/2)(v+x^2)^{(v+1)/2}} \sum_{i=0}^{\infty} \Gamma((v+i+1)/2)\left(\frac{\delta^i}{i!}\right)\left(\frac{2x^2}{v+x^2}\right)^{i/2} dx$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to t_0 is P.

Example

This example finds the probability that a *t* random variable with six degrees of freedom is greater in absolute value than 2.447. Argument *t* is symmetric about zero.

```
p = 2 * TCDF(-2.447, 6)
PM, 'Pr(|t(6)| > 2.447) = ', p, $
   Format = '(a21, f7.4)'
Pr(|t(6)| > 2.447) =  0.0500
```

Informational Errors

`STAT_OVERFLOW` — Function TCDF is set to machine infinity since overflow would occur upon modifying the inverse value for the *F* distribution with the result obtained from the inverse beta distribution.

GAMMACDF Function

Evaluates the gamma distribution function.

Usage

result = GAMMACDF(*x*, *a*)

Input Parameters

x — Argument for which the gamma distribution function is to be evaluated.

a — Shape parameter of the gamma distribution. This parameter must be positive.

Returned Value

result — The probability that a gamma random variable takes a value less than or equal to *x*.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function GAMMACDF evaluates the distribution function, *F*, of a gamma random variable with shape parameter *a*; that is,

$$F(x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. (The gamma function is the integral from 0 to *infinity* of the same integrand as above.) The value of the distribution function at the point *x* is the probability that the random variable takes a value less than or equal to *x*.

The gamma distribution is often defined as a two-parameter distribution with a scale parameter *b* (which must be positive) or even as a three-parameter distri-

bution in which the third parameter *c* is a location parameter. In the most general case, the probability density function over (*c*, *infinity*) is as follows:

$$f(t) = \frac{1}{b^a \Gamma(a)} e^{-(t-c)/b} (x-c)^{a-1}$$

If *T* is such a random variable with parameters *a*, *b*, and *c*, the probability that $T \le t_0$ can be obtained from GAMMACDF by setting $x = (t_0 - c) / b$.

If *x* is less than *a* or if *x* is less than or equal to 1.0, GAMMACDF uses a series expansion; otherwise, a continued fraction expansion is used. (See Abramowitz and Stegun, 1964.)

Example

Let *X* be a gamma random variable with a shape parameter of 4. (In this case, it has an Erlang distribution, since the shape parameter is an integer.) This example finds the probability that *X* is less than 0.5 and the probability that *X* is between 0.5 and 1.0.

```
a = 4
x = .5
p = GAMMACDF(x, a)
PM, p, Title = $
   'The probability that X is less ' + $
   'than .5 is:'
The probability that X is less than .5 is:
   0.00175162
x = 1
p = GAMMACDF(x, a) - p
PM, p, Title = $
   'The probability that X is ' + $
   'between .5 and 1 is:'
The probability that X is between .5 and 1
   is: 0.0172365
```

Informational Errors

STAT_LESS_THAN_ZERO — Input argument, *x*, is less than zero.

Fatal Errors

STAT_X_AND_A_TOO_LARGE — Function overflows because *x* and *a* are too large.

BETACDF Function

Evaluates the beta probability distribution function.

Usage

result = BETACDF(*x*, *pin*, *qin*)

Input Paramters

x — Argument for which the beta probability distribution function is to be evaluated.

pin — First beta distribution parameter. Argument *pin* must be positive.

qin — Second beta distribution parameter. Argument *qin* must be positive.

Returned Value

result — The probability that a beta random variable takes on a value less than or equal to *x*.

Input Keywords

Double — If present and nonzero, double precision is used.

Inverse — If present and nonzero, evaluates the inverse of the Beta distribution function. If *Inverse* is specified, argument *x* represents the probability for which the inverse of the Beta distribution function is to be evaluated. In this case, *x* must be in the open interval (0.0, 1.0).

Discussion

Function BETACDF evaluates the distribution function of a beta random variable with parameters *pin* and *qin*. This function is sometimes called the

incomplete beta ratio and is denoted by $I_x(p, q)$, where $p = pin$ and $q = qin$. It is given by

$$I_x(p, q) = \frac{\Gamma(p)\Gamma(q)}{\Gamma(p+q)} \int_0^x t^{p-1}(1-t)^{q-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function by $I_x(p, q)$ is the probability that the random variable takes a value less than or equal to x.

The integral in the expression above is called the incomplete beta function and is denoted by $\beta_x(p, q)$. The constant in the expression is the reciprocal of the beta function (the incomplete function evaluated at 1) and is denoted by $\beta_x(p, q)$.

If the keyword *Inverse* is specified, the BETACDF function evaluates the inverse distribution function of a beta random variable with parameters *pin* and *qin*. With $P = x$, $p = pin$ and $q = qin$, it returns x such that

$$P = \frac{\Gamma(p)\Gamma(q)}{\Gamma(p+q)} \int_0^x t^{p-1}(1-t)^{q-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to x is P.

The BETCDF function uses the method of Bosten and Battiste (1974).

Example

Suppose X is a beta random variable with parameters 12 and 12 (X has a symmetric distribution). This example finds the probability that X is less than 0.6 and the probability that X is between 0.5 and 0.6. (Since X is a symmetric beta random variable, the probability that it is less than 0.5 is 0.5.)

```
p = BETACDF(.6, 12, 12)
    ; Call BETACDF to compute the first probability
    ; and output the results.
PM, p, Title = $
    'The probability that X is less than ' + $
    '0.6 is:', Format= '(f8.4)'
```

```
The probability that X is less than 0.6 is:
    0.8364
```

```
p = p - BETACDF(.5, 12, 12)
    ; Call BETACDF and use the previously computed
    ; probability to determine the next probability.
PM, p, Format = '(f8.4)', $
   title = 'The  probability that X ' + $
   'is between 0.5 and 0.6 is:'
The probability that X is between 0.5 and 0.6
   is: 0.3364
```

BINOMIALCDF Function

Evaluates the binomial distribution function.

Usage

result = BINOMIALCDF(*k*, *n*, *p*)

Input Parameters

k — Argument for which the binomial distribution function is to be evaluated.

n — Number of Bernoulli trials.

p — Probability of success on each trial.

Returned Value

result — The probability that *k* or fewer successes occur in *n* independent Bernoulli trials, each of which has a probability *p* of success.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function BINOMIALCDF evaluates the distribution function of a binomial random variable with parameters *n* and *p* by summing probabilities of the random

variable taking on the specific values in its range. These probabilities are computed by the following recursive relationship:

$$Pr(X = j) = \frac{(n + 1 - j)p}{j(1 - p)} Pr(X = j - 1)$$

To avoid the possibility of underflow, the probabilities are computed forward from 0 if k is not greater than n times p; otherwise, they are computed backward from n. The smallest positive machine number, ε, is used as the starting value for summing the probabilities, which are rescaled by $(1 - p)^n\varepsilon$ if forward computation is performed and by $p^n\varepsilon$ if backward computation is done.

For the special case of $p = 0$, BINOMIALCDF is set to 1; for the case $p = 1$, BINOMIALCDF is set to 1 if $k = n$ and is set to zero otherwise.

Example

Suppose X is a binomial random variable with $n = 5$ and $p = 0.95$. This example finds the probability that X is less than or equal to 3.

```
p = BINOMIALCDF(3, 5, .95)

PM, 'Pr(x < 3) = ', p, $
   Format = '(a12, f7.4)'

Pr(x < 3) =  0.0226
```

Informational Errors

STAT_LESS_THAN_ZERO — Input parameter, k, is less than zero.

STAT_GREATER_THAN_N — Input parameter, k, is greater than the number of Bernoulli trials, n.

HYPERGEOCDF Function

Evaluates the hypergeometric distribution function.

Usage

result = HYPERGEOCDF(*k*, *n*, *m*, *l*)

Input Parameters

k — Parameter for which the hypergeometric distribution function is to be evaluated.

n — Sample size. Argument *n* must be greater than or equal to *k*.

m — Number of defectives in the lot.

l — Lot size. Parameter *l* must be greater than or equal to *n* and *m*.

Returned Value

result — The probability that *k* or fewer defectives occur in a sample of size *n* drawn from a lot of size *l* that contains *m* defectives.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function HYPERGEOCDF evaluates the distribution function of a hypergeometric random variable with parameters *n*, *l*, and *m*. The hypergeometric random variable *X* can be thought of as the number of items of a given type in a random sample of size *n* that is drawn without replacement from a population of size *l* containing *m* items of this type.

The probability function is

$$Pr(x = j) = \frac{\binom{m}{j}\binom{l-m}{n-j}}{\binom{l}{n}} \qquad \text{for } j = i, i+1, \ldots, \min(n, m)$$

where $i = \max(0, n - l + m)$.

If *k* is greater than or equal to *i* and less than or equal to min(*n*, *m*), BINOMIALCDF sums the terms in this expression for *j* going from *i* up to *k*; otherwise, 0 or 1 is returned, as appropriate. To avoid rounding in the accumulation, BINOMIALCDF performs the summation differently, depending on whether or not *k* is greater than the mode of the distribution, which is the greatest integer in (*m* + 1) (*n* + *1) / (l* + 2).

Example

Suppose *X* is a hypergeometric random variable with *n* = 100, *l* = 1000, and *m* = 70. In this example, the distribution function is evaluated at 7.

```
p = HYPERGEOCDF(7, 100, 70, 1000)
PM, 'Pr(x <= 7) = ', p, $
   Format = '(a13,f7.4)'
Pr(x <= 7) =  0.5995
```

Informational Errors

STAT_LESS_THAN_ZERO — Input parameter, *k*, is less than zero.

STAT_K_GREATER_THAN_N — Input parameter, *k*, is greater than the sample size.

Fatal Errors

STAT_LOT_SIZE_TOO_SMALL — Lot size must be greater than or equal to *n* and *m*.

POISSONCDF Function

Evaluates the Poisson distribution function.

Usage

result = POISSONCDF(*k*, *theta*)

Input Parameters

k — Parameter for which the Poisson distribution function is to be evaluated.

theta — Mean of the Poisson distribution. Parameter *theta* must be positive.

Returned Value

result — The probability that a Poisson random variable takes a value less than or equal to *k*.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function POISSONCDF evaluates the distribution function of a Poisson random variable with parameter *theta*. The mean of the Poisson random variable, *theta*, must be positive.

The probability function (with θ = *theta*) is as follows:

$$f(x) = (e^{-\theta}\theta^{x})/x! \quad \text{for } x = 0, 1, 2, \ldots$$

The individual terms are calculated from the tails of the distribution to the mode of the distribution and summed. The POISSONCDF function uses the recursive relationship

$$f(x+1) = f(x)(\theta/(x+1)) \quad , \qquad \text{for } x = 0, 1, 2, \ldots, k-1$$

with $f(0) = e^{-\theta}$.

Example

Suppose X is a Poisson random variable with $\theta = 10$. This example evaluates the probability that $X \leq 7$.

```
p = POISSONCDF(7, 10)
PM, 'Pr(x <= 7) = ', p, $
   Format = '(a13,f7.4)'
Pr(x <= 7) =  0.2202
```

Informational Errors

STAT_LESS_THAN_ZERO — Input parameter, *k*, is less than zero.

12

CHAPTER

Utilities

Contents of Chapter

DAYSTODATE Procedure

Gives the date corresponding to the number of days since January 1, 1900.

Usage

DAYSTODATE, *days*, *day* [, *month* [, *year*]]

Input Parameters

days — Number of days since January 1, 1900.

Output Parameters

day — On return, this named variable is assigned the day of the date specified by *days*.

month — If present, on return, this named variable is assigned the month of the date specified by *days*.

year — If present, on return, this named variable is assigned the year of the date specified by *days*. The year 1950 corresponds to the year 1950 A.D., and the year 50 corresponds to year 50 A.D.

Discussion

Procedure DAYSTODATE computes the date corresponding to the number of days since January 1, 1900. For a negative input value of *days*, the date computed is prior to January 1, 1900. This procedure is the inverse of PV-WAVE:IMSL Mathematics function DATETODAYS (page 559).

The beginning of the Gregorian calendar was the first day after October 4, 1502, which became October 15, 1582. Prior to that, the Julian calendar was in use.

Example

The following example uses DAYSTODATE to compute the date for the 100th day of 1986. This is accomplished by first using function DATETODAYS (page 559) to get the "day number" for December 31, 1985.

```
d0 = DATETODAYS(31, 12, 1985)
DAYSTODATE, d0 + 100, d, m, y
```

```
PM, d, m, y, Title = $
   'Day 100 of 1986 is (day-month-year)', $
   Format = '(20x, i3, i4, i7)'
Day 100 of 1986 is (day-month-year)
10   4   1986
```

DATETODAYS Function

Computes the number of days from January 1, 1900, to the given date.

Usage

result = DATETODAYS([*day* [, *month* [, *year*]]])

Input Parameters

day — Day of the input date.

month — Month of the input date.

year — Year of the input date. The year 1950 corresponds to the year 1950 A.D., and the year 50 corresponds to year 50 A.D.

Returned Value

result — Number of days from January 1, 1900, to the given date. If negative, it indicates the number of days prior to January 1, 1900.

Discussion

Function DATETODAYS returns the number of days from January 1, 1900, to the given date and returns negative values for days prior to January 1, 1900. A negative *year* can be used to specify B.C. Input dates in year 0 and for October 5, 1582, through October 14, 1582, inclusive, do not exist; consequently, in these cases, DATETODAYS issues an error.

The beginning of the Gregorian calendar was the first day after October 4, 1582, which became October 15, 1582. Prior to that, the Julian calendar was in use.

Example

The following example uses DATETODAYS to compute the number of days from January 15, 1986, to February 28, 1986.

```
d0 = DATETODAYS(15, 1, 1986)
d1 = DATETODAYS(28, 2, 1986)
PM, d1 - d0, Title = $
    'Number of days from 1/15/86 to 2/28/86'
Number of days from 1/15/86 to 2/28/86
 44
```

CONSTANT Function

Returns the value of various mathematical and physical constants.

Usage

result = CONSTANT(*name* [, *units*])

Input Parameters

name — Scalar string specifying the name of the desired constant. The case of the characters is not relevant when specifying *name*, i.e., character strings "PI", "Pi", "pI", and "pi" are equivalent. Spaces and underscores are allowed and ignored.

units — Scalar string specifying the units of the desired constant. If empty, then Systeme International d'Unites (SI) units are assumed. The case of the characters is not relevant when specifying *units*, i.e., character strings "METER", "Meter", and "meter" are equivalent. Parameter *units* has the form "U1*U2*...*Um/V1/.../Vn," where U_i and V_i are the names of basic units or the names of basic units raised to a power. Basic units must be separated by * or /. Powers are indicated by ^, as in "m^2" for m^2. Examples are "METER*KILOGRAM/SECOND", "M*KG/S", "METER", or "M/KG^2".

Returned Value

result — By default, returns the desired constant. If no value can be computed, NaN (Not a Number) is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

The names allowed are listed in the following table. Values marked with (mp) are exact (to machine precision). The references in the right-hand column are indicated by code numbers: (1) for Cohen and Taylor (1986), (2) for Liepman (1964), and (3) for precomputed mathematical constants.

Name	Description	Value	Reference
amu	atomic mass unit	$1.6605655 \times 10^{-27}$ kg	1
ATM	standard atm. pressure	1.01325×10^{5} N/m^2 (mp)	2
AU	astronomical unit	1.496×10^{11} m	
Avogadro	Avogadro's number, *N*	6.022045×10^{23} 1/mole	1
Boltzman	Boltzman's constant, *k*	1.380662×10^{-23} J / K	1
C	speed of light, *c*	$2.997924580 \times 10^{8}$ m/s	1
Catalan	Catalan's constant	0.915965... (mp)	3
E	base of natural logs, *e*	2.718... (mp)	3
ElectronCharge	electron charge, *e*	$1.6021892 \times 10^{-19}$ C	1
ElectronMass	electron mass, m_e	9.109534×10^{-31} kg	1
ElectronVolt	electron volt, *ev*	$1.6021892 \times 10^{-19}$ J	1
Euler	Euler's constant, γ	0.577... (mp)	3
Faraday	Faraday constant, *F*	9.648456×10^{4} C/mole	1
FineStructure	fine structure, α	7.2973506×10^{-3}	1
Gamma	Euler's constant, γ	0.577... (mp)	3
Gas	gas constant, R_0	8.31441 J/mole/K	1
Gravity	gravitational constant, *G*	6.6720×10^{-11} N m^2 / kg^2	1
Hbar	Planck's constant / 2π	$1.0545887 \times 10^{-34}$ J s	1
PerfectGasVolume	std. vol. ideal gas	2.241383×10^{-2} m^3 / mole	1

Name	Description	Value	Reference
Pi	Pi, π	3.141... (mp)	3
Planck	Planck's constant, *h*	6.626176×10^{-34} J s	1
ProtonMass	proton mass, M_p	$1.6726485 \times 10^{-27}$ kg	1
Rydberg	Rydberg's constant, R_{infinity}	$1.097373177 \times 10^{7}$ /m	1
Speedlight	speed of light, *c*	$2.997924580 \times 10^{8}$ m/s	1
StandardGravity	standard *g*	9.80665 m/s^2 (mp)	2
StandardPressure	standard atm. pressure	1.01325×10^{5} N/m^2 (mp)	2
StefanBoltzman	Stefan-Boltzman, σ	5.67032×10^{-8} W/K^4 $/m^2$	1
WaterTriple	triple point of water	2.7316×10^{2} K	2

The units allowed are as follows:

Unit	Description
time	day, hour = hr, min = minute, s = sec = second, year
frequency	Hertz = Hz
mass	AMU, g = gram, lb = pound, ounce = oz, slug
distance	Angstrom, AU, feet = foot, in = inch, m = meter = metre, micron, mile, mill, parsec, yard
area	acre
volume	l = liter = litre
force	dyne, N = Newton
energy	BTU, Erg, J = Joule
work	W = watt
pressure	ATM = atmosphere, bar
temperature	degC = Celsius, degF = Fahrenheit, degK = Kelvin
viscosity	poise, stoke
charge	Abcoulomb, C = Coulomb, statcoulomb
current	A = ampere, abampere, statampere

Unit	Description
voltage	Abvolt, V = volt
magnetic induction	T = Tesla, Wb = Weber
other units	l, farad, mole, Gauss, Henry, Maxwell, Ohm

The following metric prefixes can be used with the above units. The one- or two-letter prefixes can only be used with one-letter unit abbreviations.

a	atto	10^{-18}
f	femto	10^{-15}
p	pico	10^{-12}
n	nano	10^{-9}
u	micro	10^{-6}
m	milli	10^{-3}
c	centi	10^{-2}
d	deci	10^{-1}
dk	deca	10^{2}
k	kilo	10^{3}
	myria	10^{4}
	mega	10^{6}
g	giga	10^{9}
t	tera	10^{12}

There is no one-letter unit abbreviation for *myria* or *mega* since *m* means *milli.*

Example 1

In this example, Euler's constant γ is obtained and printed. Euler's constant is defined to be as follows:

$$\gamma = \lim_{n \to \infty} \left[\sum^{n-1} \frac{1}{k} - \ln n \right]$$

```
PM, CONSTANT('gamma')
0.577216
```

Example 2

In this example, the speed of light is obtained using several different units.

```
c1 = CONSTANT('SpeedLight', 'meter/second')
c2 = CONSTANT('SpeedLight', 'mile/second')
c3 = CONSTANT('SpeedLight', 'cm/ns')
PM, 'speed of light = ', c1, c2, c3, $
   Title ='               meters/second    ' + $
   'miles/second      cm/ns'
 meters/second  miles/second  cm/ns
speed of light = 2.99792e+08     186282.       29.9793
```

Warning Errors

MATH_MASS_TO_FORCE — Conversion of units-of-mass to units-of-force required for consistency.

MACHINE Function

Returns information describing the computer's arithmetic.

Usage

result = MACHINE( )

Returned Value

result — The information describing the computer's arithmetic is returned in a structure.

Output Keywords

Float — If present and nonzero, a structure containing the information describing the single-precision, floating-point arithmetic is returned.

Double — If present and nonzero, a structure containing the information describing the single-precision, floating-point arithmetic is returned.

Discussion

Function MACHINE returns information describing the computer's arithmetic. This can be used to make programs machine independent. The information returned by MACHINE is in the form of a structure. A different structure is used for each type: integer, float, and double. Depending on how MACHINE is called, a different structure is returned.

The default action of MACHINE is to return the structure IMACHINE which contains integer information on the computer's arithmetic. By using either the keywords *Float* or *Double*, information about the floating- or double-precision arithmetic is returned in structures FMACHINE or DMACHINE.

The contents of the these structures are described below.

Integer Information: IMACHINE

Assume that integers are represented in *M*-digit, base *A* form as

$$\sigma \sum^{M} x_k A^k$$

where σ is the sign and $0 \le x_k < A$ for $k = 0, ..., M$. Then, the following table describes the tags:

Tag	Definition
BITS_PER_CHAR	*C*, bits per character
INTEGER_BASE	*A*, the base
INTEGER_DIGITS	M_S, the number of base-*A* digits in a *short int*
MAX_INTEGER	$A^{M_s} - 1$, the largest *short int*
LONG_DIGITS	M_l, the number of base-A digits in a *long int*
MAX_LONG	$A^{M_l} - 1$, the largest *long int*

Assume that floating-point numbers are represented in *N*-digit, base *B* form as

$$\sigma B^E \sum^{N} x_k B^{-k}$$

where σ is the sign and $0 \le x_k < B$ for $k = 1, ..., N$ for and $E_{\min} \le E \le E_{\max}$.

Tag	Definition
FLOAT_BASE	*B*, the base
FLOAT_DIGITS	N_f, the number of base-*B* digits in *float*
FLOAT_MIN_EXP	$E_{\min_f}$, the smallest *float* exponent
FLOAT_MAX_EXP	$E_{\max_f}$, the largest *float* exponent
DOUBLE_DIGETS	N_d, the number of base-*B* digits in *double*
DOUBLE_MIN_EXP	$E_{\min_d}$, the largest *long int*
DOUBLE_MAX_EXP	$E_{\max_d}$, the number of base-*B* digits in *double*

Floating- and Double-precision Information: FMACHINE and DMACHINE

Information concerning the floating- or double-precision arithmetic of the computer is contained in the structures FMACHINE and DMACHINE. These structures are returned into named variables by calling MACHINE with the keywords *Float* for FMACHINE and *Double* for DMACHINE.

Assume that *float* numbers are represented in N_f- digit, base B form as

$$\sigma B^E \sum_{k=1}^{N} x_k B^{-k},$$

where σ is the sign, $0 \le x_k < B$ for $k = 1, 2, ..., N_f$ and

$$E_{\min_f} \le E \le E_{\max_f}.$$

Note that if we make the assignment imach = MACHINE( ), then B = imach.FLOAT_BASE, N_f = imach.FLOAT_DIGITS,

$$E_{\min_f} = \text{imach.FLOAT\_MIN\_EXP},$$

and

$$E_{\max_f} = \text{imach.FLOAT\_MAX\_EXP}.$$

The ANSI/IEEE 754-1985 standard for binary arithmetic uses NaN (Not a Number) as the result of various otherwise illegal operations, such as computing 0/0. If the assignment amach = MACHINE(/*Float*) is made, then on computers that do not support NaN, a value larger than amach. MAX_POS is returned in amach.NAN. On computers that do not have a special representation for infinity, amach.POS_INF contains the same value as amach.MAX_POS.

The structure IMACHINE is defined by the following table:

Tag	**Definition**
MIN_POS	$B^{Emin}f^{-1}$, the smallest positive number
MAX_POS	$B^{Emax}f(1 - B^{-Nf})$, the largest number
MIN_REL_SPACE	$B - N_f$, the smallest relative spacing
MAX_REL_SPACE	$B^1 - N_f$, the largest relative spacing
LOG10_BASE	$\log_{10}(B)$
NAN	NaN
POS_INF	positive machine infinity
NEG_INF	negative machine infinity

The structure DMACHINE contains machine constants that define the computer's double arithmetic. Note that for *double,* if the assignment imach = MACHINE( ) is made, then

B = *imach.FLOAT_BASE*, N_f = *imach.DOUBLE_DIGITS*,

$E_{\min_f}$ = imach.DOUBLE_MIN_EXP,

and

$E_{\max_f}$ = imach.DOUBLE_MAX_EXP.

Missing values in PV-WAVE:IMSL Mathematics procedures and functions are often indicated by NaN. There is no missing-value indicator for integers. Users ususally have to convert from their missing value indicators to NaN.

Example

In this example, all values returned by MACHINE are printed on a machine with IEEE (Institute for Electrical and Electronics Engineering) arithmetic.

```
i = machine()
f = machine(/Float)
d = machine(/Double)
    ; Call INFO with the keyword Structure set to view the contents of the structures.
INFO, i, f, d, /Structure
** Structure IMACHINE, 13 tags, length=52:
               BITS_PER_CHAR      LONG                      8
               INTEGER_BASE       LONG                      2
               INTEGER_DIGITS     LONG                     15
               MAX_INTEGER        LONG                  32767
               LONG_DIGITS        LONG                     31
               MAX_LONG           LONG            2147483647
               FLOAT_BASE         LONG                      2
               FLOAT_DIGITS       LONG                     24
               FLOAT_MIN_EXP      LONG                   -125
               FLOAT_MAX_EXP      LONG                    128
               DOUBLE_DIGITS      LONG                     53
               DOUBLE_MIN_EXP     LONG                  -1021
               DOUBLE_MAX_EXP     LONG                   1024
** Structure FMACHINE, 8 tags, length=32:
```

```
                MIN_POS           FLOAT         1.17549e-38
                MAX_POS           FLOAT         3.40282e+38
                MIN_REL_SPACE     FLOAT         5.96046e-08
                MAX_REL_SPACE     FLOAT         1.19209e-07
                LOG_10            FLOAT             0.301030
                NAN               FLOAT                  NaN
                POS_INF           FLOAT                  Inf
                NEG_INF           FLOAT                 -Inf
** Structure DMACHINE, 8 tags, length=64:
                MIN_POS           DOUBLE    2.2250739e-308
                MAX_POS           DOUBLE    1.7976931e+308
                MIN_REL_SPACE     DOUBLE     1.1102230e-16
                MAX_REL_SPACE     DOUBLE     2.2204460e-16
                LOG_10            DOUBLE         0.30102998
                NAN               DOUBLE                 NaN
                POS_INF           DOUBLE           Infinity
                NEG_INF           DOUBLE          -Infinity
```

NORM Function

Computes various norms of a vector or the difference of two vectors.

Usage

result = NORM(*x* [, *y*])

Input Parameters

x — Vector for which the norm is to be computed.

y — If present, NORM computes the norm of $(x - y)$.

Returned Value

result — The requested norm of the input vector. If the norm cannot be computed, NaN is returned.

Input Keywords

One — If present and nonzero, computes the 1-norm

$$\sum_{i=0}^{n-1} |x_i| .$$

Inf — If present and nonzero, computes the infinity norm $\max|x_i|$.

Output Keywords

Index_Max — Named variable into which the index of the element of *x* with the maximum modulus is stored. If *Index_Max* is used, then the keyword *Inf* also must be used. If the parameter *y* is specified, then the index of $(x - y)$ with the maximum modulus is stored.

Discussion

By default, NORM computes the Euclidean norm as follows:

$$\left(\sum_{i=0}^{n-1} x_i^2 \right)^{\frac{1}{2}}$$

If the keyword *One* is set, then the 1-norm

$$\sum_{i=0}^{n-1} |x_i|$$

is returned. If the keyword *Inf* is set, the infinity norm $\max|x_i|$

is returned. In the case of the infinity norm, the index of the element with maximum modulus also is returned.

If the parameter *y* is specified, the computations of the norms described above are performed on $(x - y)$.

Example 1

In this example, the Euclidean norm of an input vector is computed.

```
x = [ 1.0, 3.0, -2.0, 4.0 ]
n = NORM(x)
PM, n, Title = 'Euclidean norm of x:'
Euclidean norm of x:
5.47723
```

Example 2

This example computes max $| x_i - y_i |$ and prints the norm and index.

```
x = [1.0, 3.0, -2.0, 4.0]
y = [4.0, 2.0, -1.0, -5.0]
n = NORM(x, y, /Inf, Index_Max = imax)
PM, n, Title = 'Infinity norm of (x-y):'
Infinity norm of (x-y):
       9.00000
```

```
PM, imax, Title = $
    'Element of (x-y) with maximum modulus:'
Element of (x-y) with maximum modulus:
3
```

MATRIX_NORM Function

Computes various norms of a rectangular matrix, a matrix stored in band format, and a matrix stored in coordinate format.

Usage

result = MATRIX_NORM(*a*)
Computes various norms of a rectangular matrix.

result = MATRIX_NORM(*n*, *nlca*, *nuca*, *a*)
Computes various norms of a matrix stored in band format.

result = MATRIX_NORM(*nrows*, *ncols*, *a*)
Computes various norms of a matrix stored in coordinate format.

Input Parameters

a — Matrix for which the norm will be computed.

n — The order of matrix *A*.

nlca — Number of lower codiagonals of *A*.

nuca — Number of upper codiagonals of *A*.

nrows — The number of rows in matrix *A*.

ncols — The number of columns in matrix *A*.

Returned Value

result — The requested norm of the input matrix, by default, the Frobenius norm. If the norm cannot be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

One_Norm — If present and nonzero, function MATRIX_NORM computes the one norm of matrix *A*.

Inf_Norm — If present and nonzero, function MATRIX_NORM computes the infinity norm of matrix *A*.

Symmetric — If present and nonzero, matrix *A* is stored in symmetric storage mode. Keyword *Symmetric* can not be used with a rectangular matrix.

Discussion

By default, MATRIX_NORM computes the Frobenius norm

$$\|A\|_2 = \left[\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} A_{ij}^2 \right]^{\frac{1}{2}}$$

If the keyword *One_Norm* is used, the one norm

$$\|A\|_1 = \max_{0 \le j \le n-1} \sum_{i=0}^{m-1} |A_{ij}|$$

is returned. If the keyword *Inf_Norm* is used, the infinity norm

$$\|A\|_\infty = \max_{0 \le i \le m-1} \sum_{j=0}^{n-1} |A_{ij}|$$

is returned.

Example 1

Compute the Frobenius norm, infinity norm, and one norm of matrix *A*.

```
a   =   TRANSPOSE([[1.0, 2.0, -2.0, 3.0], $
                   [-2.0, 1.0, 3.0, 0.0], $
                   [0.0, 3.0, 1.0, -7.0], $
                   [5.0, -2.0, 7.0, 6.0], $
                   [4.0, 3.0, 4.0, 0.0]])
```

```
frobenius_norm  =  MATRIX_NORM(a)
inf_norm  =  MATRIX_NORM(a, /Inf_Norm)
one_norm  =  MATRIX_NORM(a, /One_Norm)
PRINT, "Frobenius norm = ", frobenius_norm
PRINT, "Infinity norm  = ", inf_norm
PRINT, "One norm        = ", one_norm
Frobenius norm =        15.6844
Infinity norm  =        20.0000
One norm        =        17.0000
```

Example 2

Compute the Frobenius norm, infinity norm, and one norm of matrix *A*. Matrix *A* is stored in band storage mode.

```
nlca  =  1
nuca  =  1
n  =  4
a  =  [0.0, 2.0, 3.0, -1.0, 1.0, 1.0, $
       1.0, 1.0, 0.0, 3.0, 4.0, 0.0]
frobenius_norm  =  MATRIX_NORM(n, nlca, nuca, a)
inf_norm  =  MATRIX_NORM(n, nlca, nuca, a, /Inf_Norm)
one_norm  =  MATRIX_NORM(n, nlca, nuca, a, /One_Norm)
PRINT, "Frobenius norm = ", frobenius_norm
PRINT, "Infinity norm  = ", inf_norm
PRINT, "One norm        = ", one_norm
Frobenius norm =        6.55744
Infinity norm  =        5.00000
One norm        =        8.00000
```

Example 3

Compute the Frobenius norm, infinity norm, and one norm of matrix *A*. Matrix *A* is stored in symmetric band storage mode.

```
nlca   =  2
nuca   =  2
n   =   6
a   =   [0.0, 0.0, 7.0, 3.0, 1.0, 4.0, $
          0.0, 5.0, 1.0, 2.0, 1.0, 2.0, $
          1.0, 2.0, 4.0, 6.0, 3.0, 1.0]
frobenius_norm   =   MATRIX_NORM(n, nlca, nuca, a, $
                                 /Symmetric)
inf_norm   =   MATRIX_NORM(n, nlca, nuca, a, /Inf_Norm, $
                                 /Symmetric)
one_norm   =   MATRIX_NORM(n, nlca, nuca, a, /One_Norm, $
                                 /Symmetric)
PRINT, "Frobenius norm = ", frobenius_norm
PRINT, "Infinity norm   = ", inf_norm
PRINT, "One norm         = ", one_norm
Frobenius norm =         16.9411
Infinity norm   =         16.0000
One norm         =          16.0000
```

Example 4

Compute the Frobenius norm, infinity norm, and one norm of matrix *A*. Matrix *A* is stored in coordinate format.

```
nrows   =  6
ncols   =  6
a   =   REPLICATE(!F_Sp_Elem, 15)
a(*).row   =   [0, 1, 1, 1, 2, $
                  3, 3, 3, 4, 4, $
                  4, 4, 5, 5, 5]
a(*).col   =   [0, 1, 2, 3, 2, $
                  0, 3, 4, 0, 3, $
                  4, 5, 0, 1, 5]
```

```
a(*).val   =   [10.0, 10.0, -3.0, -1.0, 15.0, $
                  -2.0, 10.0, -1.0, -1.0, -5.0, $
                  1.0, -3.0, -1.0, -2.0, 6.0]
frobenius_norm   =   MATRIX_NORM(nrows, ncols, a)
inf_norm   =   MATRIX_NORM(nrows, ncols, a, /Inf_Norm)
one_norm   =   MATRIX_NORM(nrows, ncols, a, /One_Norm)
PRINT, "Frobenius norm = ", frobenius_norm
PRINT, "Infinity norm   = ", inf_norm
PRINT, "One norm         = ", one_norm
Frobenius norm =         24.8395
Infinity norm   =         15.0000
One norm         =         18.0000
```

Example 5

Compute the Frobenius norm, infinity norm and one norm of matrix *A*. Matrix *A* is stored in symmetric coordinate format.

```
nrows   =   6
ncols   =   6
a   =   REPLICATE(!F_Sp_Elem, 9)
a(*).row   =   [0, 0, 0, $
                  1, 1, 2, $
                  2, 4, 4]
a(*).col   =   [0, 2, 5, $
                  3, 4, 2, $
                  5, 4, 5]
a(*).val   =   [10.0, -1.0, 5.0, $
                  2.0, 3.0, 3.0, $
                  4.0, -1.0, 4.0]
frobenius_norm   =   MATRIX_NORM(nrows, ncols, a, $
                                          /Symmetric)
inf_norm   =   MATRIX_NORM(nrows, ncols, a, /Inf_Norm, $
                                          /Symmetric)
one_norm   =   MATRIX_NORM(nrows, ncols, a, /One_Norm, $
                                          /Symmetric)
```

```
PRINT, "Frobenius norm = ", frobenius_norm
PRINT, "Infinity norm   = ", inf_norm
PRINT, "One norm         = ", one_norm
Frobenius norm =         15.8745
Infinity norm   =         16.0000
One norm         =         16.0000
```

CMAST_ERR_STOP Function

Options for error recovery in Math and Stat options.

Usage

CMAST_ERR_STOP, *lev*

Input Parameters

lev — Integer specifying the stopping level.

Discussion

Function CMAST_ERR_STOP allows users to define how the Math and Stat options will behave when a Terminal or Fatal error occurs. Setting *lev* to one will force the Math/Stat routine to stop execution when a Terminal or Fatal error occurs (default). Setting *lev* to zero will force the Math/Stat routine to continue execution when a Terminal or Fatal error occurs.

CMAST_ERR_PRINT Procedure

Sets options for error printing in Math and Stat options.

Usage

CMAST_ERR_PRINT, *lev*

Input Parameters

lev — Integer specifying the printing level.

Discussion

Function CMAST_ERR_PRINT allows users to define how the Math and Stat options will behave when an error occurs. Setting *lev* to two will force the Math/Stat routine to print all error messages that occur (default). Setting *lev* to one will force the Math/Stat routine to print only Terminal and Fatal error messages that occur. Setting *lev* to zero will force the Math/Stat to not print any error messages.

Example

See Example 1 for MINCONGEN (Chapter 8, *Optimization*).

CMAST_ERR_TRANS Function

Determines if an *Informational Error* has occurred.

Usage

result = CMAST_ERR_TRANS(*arg*)

Output Parameters

arg — Can be either a scalar string specifying a particular *Informational Error* or an integer specifying the internal code of an *Informational Error.*

Returned Value

result — If *arg* is a scalar string specifying a valid *Informational Error*, then the return value is the integer error-code value of the *Informational Error.* If *arg* is an integer specifying a valid *Informational Error* code, then a string specifying the *Informational Error* is returned.

Discussion

Function CMAST_ERROR_TRANS is designed to check programs for specific *Informational Errors*. PV-WAVE:IMSL Mathematics mathematical functions attempt to detect user errors and handle them in a way that provides as much information to the user as possible. To do this, five levels of *Informational Error* severity, in addition to the basic PV-WAVE:IMSL Mathematics error-handling facility, are recognized. Following a call to a mathematical or statistical function, the system variables !Error and !Cmast_Err contain information concerning the current error state. Variable !Error contains the error *number* of the last error, and !Cmast_Err is set either to zero, which indicates that an *Informational Error* did not occur, or to the error *code* of the last *Informational Error* that did occur.

The user can interact with the PV-WAVE:IMSL Mathematics error-handling system with respect to *Informational Errors* in two ways: (1) change the default printing actions and (2) determine the code of an *Informational Error* so as to take corrective action. To change the default printing action, the system variable !Quiet is set to a nonzero value. To allow for corrective action to be taken based on the existence of a particular *Informational Error*, function

CMAST_ERR_TRANS retrieves the integer code for an *Informational Error* given a scalar string specifying the name given to the error.

In the program segment below, the Cholesky factorization of a matrix is to be performed. If it is determined that the matrix is not nonnegative definite (and often this is not immediately obvious), the program is to take a different branch.

```
x = CHNNDFAC, a, fac
    ; Call CHNNDFAC with a matrix that may not be nonnegative definite.
IF (CMAST_ERROR_TRANS($
    'MATH_NOT_NONNNEG_DEFINITE') EQ $
    !Cmast_Err))$
        ; Check the system variable Cmast_Err to see if it contains the
        ; error code for the error
        ; MATH_NOT_NONNNEG_DEFINITE.
THEN ;... Handle matrix that is not nonnegative definite.
```

A

APPENDIX

References

Abramowitz, Milton, and Irene A. Stegun (editors) (1964), *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, National Bureau of Standards, Washington, D.C.

Afifi, A.A., and S.P. Azen (1979), *Statistical Analysis: A Computer Oriented Approach*, 2d ed., Academic Press, New York.

Ahrens, J.H., and U. Dieter (1974), Computer methods for sampling from gamma, beta, Poisson, and binomial distributions, *Computing*, **12**, 223–246.

Akaike, H. (1978), A Bayesian analysis of the minimum AIC procedure, *Ann. Institute Statist. Mathematics.*, **30A**, 9–14.

Akaike, H. (1973), Information theory and an extension of maximum likelihood principle, *Proc. 2nd International Symposium on Information Theory*, Eds. B.N. Petrov and F. Csaki, 267–281.

Akima, H. (1978), A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points, *ACM Transactions on Mathematical Software*, **4**, 148–159.

Akima, H. (1970), A new method of interpolation and smooth curve fitting based on local procedures, *Journal of the ACM*, **17**, 589–602.

Anderson, R.L., and T.A. Bancroft (1952), *Statistical Theory in Research*, McGraw-Hill Book Company, New York.

Anderson, T.W. (1971), *The Statistical Analysis of Time Series*, John Wiley & Sons, New York.

Ashcraft, C. (1987), *A vector implementation of the multifrontal method for large sparse symmetric positive definite systems*, Technical Report ETA-TR-51, Engineering Technology Applications Division, Boeing Computer Services, Seattle, Washington.

Ashcraft, C., R. Grimes, J. Lewis, B. Peyton, and H. Simon (1987), Progress in sparse matrix methods for large linear systems on vector supercomputers. *Intern. J. Supercomputer Applic.*, **1**(**4**), 10-29.

Atkinson, A.C. (1979), A family of switching algorithms for the computer generation of beta random variates, *Biometrika*, **66**, 141–145.

Atkinson, A.C. (1985), *Plots, Transformations, and Regression*, Claredon Press, Oxford.

Atkinson, Ken (1978), *An Introduction to Numerical Analysis*, John Wiley & Sons, New York.

Barnett, A.R. (1981), An algorithm for regular and irregular Coulomb and Bessel functions of real order to machine accuracy, *Computer Physics Communication*, **21,** 297–314.

Barrett, J.C., and M.J.R. Healy (1978), A remark on Algorithm AS 6: Triangular decomposition of a symmetric matrix, *Applied Statistics*, **27**, 379–380.

Bays, Carter, and S.D. Durham (1976), Improving a poor random number generator, *ACM Transactions on Mathematical Software*, **2**, 59–64.

Bishop, Yvonne M.M., Stephen E. Fienberg, and Paul W. Holland (1975), *Discrete Multivariate Analysis: Theory and Practice*, MIT Press, Cambridge, Mass.

Blom, Gunnar (1958), *Statistical Estimates and Transformed Beta-Variables*, John Wiley & Sons, New York.

de Boor, Carl (1978), *A Practical Guide to Splines*, Springer-Verlag, New York.

Bosten, Nancy E., and E.L. Battiste (1974), Incomplete beta ratio, *Communications of the ACM*, **17**, 156–157.

Box, George E.P., and Gwilyn M. Jenkins (1976), *Time Series Analysis: Forecasting and Control*, revised ed., Holden-Day, Oakland.

Box, G.E.P., and P.W. Tidwell (1962), Transformation of the independent variables, *Technometrics*, **4**, 531–550.

Brent, Richard P. (1973), *Algorithms for Minimization without Derivatives*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Brigham, E. Oran (1974), *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, New Jersey.

Brown, Morton B., and Jacqualine K. Benedetti (1977), Sampling behavior and tests for correlation in two-way contingency tables, *Journal of the American Statistical Association*, **42**, 309–315.

Brown, Morton E. (1983), MCDP4F, two-way and multiway frequency tables—measures of association and the log-linear model (complete and incomplete tables), in *BMDP Statistical Software, 1983 Printing with Additions*, (edited by W.J. Dixon), University of California Press, Berkeley.

Carlson, R.E., and T.A. Foley (1991),The parameter R^2 in multiquadric interpolation, *Computer Mathematical Applications*, **21**, 29–42.

Cheng, R.C.H. (1978), Generating beta variates with nonintegral shape parameters, *Communications of the ACM*, **21**, 317–322.

Cohen, E. Richard, and Barry N. Taylor (1986), *The 1986 Adjustment of the Fundamental Physical Constants*, Codata Bulletin, Pergamon Press, New York.

Conover, W.J. (1980), *Practical Nonparametric Statistics*, 2d ed., John Wiley & Sons, New York.

Conover, W.J., and Ronald L. Iman (1983), *Introduction to Modern Business Statistics*, John Wiley & Sons, New York.

Cook, R. Dennis, and Sanford Weisberg (1982), *Residuals and Influence in Regression*, Chapman and Hall, New York.

Cooley, J.W., and J.W. Tukey (1965), An algorithm for the machine computation of complex Fourier series, *Mathematics of Computation*, **19**, 297–301.

Cooper, B.E. (1968), Algorithm AS4, An auxiliary function for distribution integrals, *Applied Statistics*, **17**, 190–192.

Craven, Peter, and Grace Wahba (1979), Smoothing noisy data with spline functions, *Numerische Mathematik*, **31**, 377–403.

Crowe, Keith, Yuan-An Fan, Jing Li, Dale Neaderhouser, and Phil Smith (1990), *A direct sparse linear equation solver using linked list storage*, IMSL Technical Report 9006, IMSL, Houston.

D'Agostino, Ralph B., and Michael A. Stevens (1986), *Goodness-of-Fit Techniques*, Marcel Dekker, New York.

Davis, Philip F., and Philip Rabinowitz (1984), *Methods of Numerical Integration*, Academic Press, Orlando, Florida.

Dallal, Gerald E. and Leland Wilkinson (1986), An analytic approximation to the distribution of Lilliefor's test statistic for normality, *The American Statistician*, **40**, 294–296.

Dennis, J.E., Jr., and Robert B. Schnabel (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Englewood Cliffs, New Jersey.

Devore, Jay L (1982), *Probability and Statistics for Engineering and Sciences*, Brooks/Cole Publishing Company, Monterey, Calif.

Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart (1979), *LINPACK User's Guide*, SIAM, Philadelphia.

Draper, N.R., and H. Smith (1981), *Applied Regression Analysis*, 2d ed., John Wiley & Sons, New York.

Du Croz, Jeremy, P. Mayes, and G. Radicati (1990), Factorization of band matrices using Level-3 BLAS, *Proceedings of CONPAR 90-VAPP IV*, Lecture Notes in Computer Science, Springer, Berlin, 222.

Duff, I. S., and J. K. Reid (1983), The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, **9**, 302-325.

Duff, I. S., and J. K. Reid (1984), The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing*, **5**, 633-641.

Duff, I. S., A. M. Erisman, and J. K. Reid (1986), *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford.

Efroymson, M.A. (1960), Multiple regression analysis, *Mathematical Methods for Digital Computers*, Volume 1, (edited by A. Ralston and H. Wilf), John Wiley & Sons, New York, 191–203.

Emmett, W.G. (1949), Factor analysis by Lawless method of maximum likelihood, *British Journal of Psychology, Statistical Section*, **2**, 90–97.

Enright, W.H., and J.D. Pryce (1987), Two FORTRAN packages for assessing initial value methods, *ACM Transactions on Mathematical Software*, **13**, 1–22.

Farebrother, R.W., and G. Berry (1974), A remark on Algorithm AS 6: Triangular decomposition of a symmetric matrix, *Applied Statistics*, **23**, 477.

Fisher, R.A. (1936), The use of multiple measurements in taxonomic problems, *The Annals of Eugenics*, **7**, 179–188.

Fishman, George S., and Louis R. Moore (1982), A statistical evaluation of multiplicative congruential random number generators with modulus $2^{31} - 1$, *Journal of the American Statistical Association*, **77**, 129–136.

Forsythe, G.E. (1957), Generation and use of orthogonal polynomials for fitting data with a digital computer, *SIAM Journal on Applied Mathematics*, **5**, 74–88.

Franke, R. (1982), Scattered data interpolation: Tests of some methods, *Mathematics of Computation*, **38**, 181–200.

Furnival, G.M. and R.W. Wilson, Jr. (1974), Regressions by leaps and bounds, *Technometrics*, **16**, 499–511.

Gautschi, Walter (1968), Construction of Gauss-Christoffel quadrature formulas, *Mathematics of Computation*, **22**, 251–270.

Gear, C.W. (1971), *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, New Jersey.

Gentleman, W. Morven (1974), Basic procedures for large, sparse or weighted linear least squares problems, *Applied Statistics*, **23**, 448–454.

George, A., and J. W. H. Liu (1981), *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

Gill, P.E., W. Murray, M.A. Saunders, and M.H. Wright (1985), Model building and practical aspects of nonlinear programming, *Computational Mathematical Programming*, (edited by K. Schittkowski), NATO ASI Series, **15**, Springer-Verlag, Berlin, Germany.

Girschick, M.A. (1939), On the sampling theory of roots of determinantal equations, *Annals of Mathematical Statistics*, **10**, 203–224.

Goldfarb, D., and A. Idnani (1983), A numerically stable dual method for solving strictly convex quadratic programs, *Mathematical Programming*, **27**, 1–33.

Golub, G.H. (1973), Some modified matrix eigenvalue problems, *SIAM Review*, **15**, 318–334.

Golub, Gene H., and Charles F. Van Loan (1983), *Matrix Computations*, Johns Hopkins University Press, Baltimore, Md.

Golub, G.H., and C.F. Van Loan (1989), *Matrix Computations*, 2d ed., The Johns Hopkins University Press, Baltimore, Maryland.

Golub, G.H., and J.H. Welsch (1969), Calculation of Gaussian quadrature rules, *Mathematics of Computation*, **23**, 221–230.

Goodnight, James H. (1979), A tutorial on the SWEEP operator, *The American Statistician*, **33**, 149–158.

Gregory, Robert, and David Karney (1969), *A Collection of Matrices for Testing Computational Algorithms*, Wiley-Interscience, John Wiley & Sons, New York.

Griffin, R., and K.A. Redish (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 54.

Grosse, Eric (1980), Tensor spline approximation, *Linear Algebra and its Applications*, **34**, 29–41.

Hageman, Louis A., and David M. Young (1981), *Applied Iterative Methods*, Academic Press, New York.

Haldane, J.B.S. (1939), The mean and variance of χ^2 when used as a test of homogeneity, when expectations are small, *Biometrika*, **31**, 346.

Hardy, R.L. (1971), Multiquadric equations of topography and other irregular surfaces, *Journal of Geophysical Research*, **76**, 1905–1915.

Harman, Harry H. (1976), *Modern Factor Analysis*, 3d ed. revised, University of Chicago Press, Chicago.

Hart, John F., E.W. Cheney, Charles L. Lawson, Hans J. Maehly, Charles K. Mesztenyi, John R. Rice, Henry G. Thacher, Jr., and Christoph Witzgall (1968), *Computer Approximations*, John Wiley & Sons, New York.

Hartigan, John A. (1975), *Clustering Algorithms*, John Wiley & Sons, New York.

Hartigan, J.A., and M.A. Wong (1979), Algorithm AS 136: A *K*-means clustering algorithm, *Applied Statistics*, **28**, 100–108.

Hayter, Anthony J. (1984), A proof of the conjecture that the Tukey-Kramer multiple comparisons procedure is conservative, *Annals of Statistics*, **12**, 61–75.

Healy, M.J.R. (1968), Algorithm AS 6: Triangular decomposition of a symmetric matrix, *Applied Statistics*, **17**, 195–197.

Hemmerle, William J. (1967), *Statistical Computations on a Digital Computer*, Blaisdell Publishing Company, Waltham, Mass.

Higham, Nicholas J. (1988), FORTRAN Codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation, *ACM Transactions on Mathematical Software*, **14**, 381-396.

Hildebrand, F.B. (1956), *Introduction to Numerical Analysis*, McGraw Hill.

Hindmarsh, A.C. (1974), *GEAR: Ordinary Differential Equation System Solver*, Lawrence Livermore National Laboratory Report UCID-30001, Revision 3, Lawrence Livermore National Laboratory, Livermore, California.

Hinkley, David (1977), On quick choice of power transformation, *Applied Statistics*, **26**, 67–69.

Hill, G.W. (1970), Student's *t*-distribution, *Communications of the ACM*, **13**, 617–619.

Hoaglin, David C., and Roy E. Welsch (1978), The hat matrix in regression and ANOVA, *The American Statistician*, **32**, 17–22.

Hocking, R.R. (1972), Criteria for selection of a subset regression: Which one should be used?, *Technometrics*, **14**, 967–970.

Huber, Peter J. (1981), *Robust Statistics*, John Wiley & Sons, New York.

Hull, T.E., W.H. Enright, and K.R. Jackson (1976), *User's Guide for DVERK–A Subroutine for Solving Nonstiff ODEs*, Department of Computer Science Technical Report 100, University of Toronto.

Irvine, Larry D., Samuel P. Marin, and Philip W. Smith (1986), Constrained interpolation and smoothing, *Constructive Approximation*, **2**, 129–151.

Jackson, K.R., W.H. Enright, and T.E. Hull (1978), A theoretical criterion for comparing Runge-Kutta formulas, *SIAM Journal of Numerical Analysis*, **15**, 618 – 641.

Jenkins, M.A. (1975), Algorithm 493: Zeros of a real polynomial, *ACM Transactions on Mathematical Software*, **1**, 178–189.

Jenkins, M.A., and J.F. Traub (1970), A three-stage algorithm for real polynomials using quadratic iteration, *SIAM Journal on Numerical Analysis*, **7**, 545–566.

John, Peter W.M. (1971), *Statistical Design and Analysis of Experiments*, Macmillan Company, New York.

Jöhnk, M.D. (1964), Erzeugung von Betaverteilten und Gammaverteilten Zufalls-zahlen, *Metrika*, **8**, 5–15.

Jöreskog, K.G. (1977), Factor analysis by least squares and maximum-likelihood methods, *Statistical Methods for Digital Computers*, (edited by Kurt Enslein, Anthony Ralston, and Herbert S. Wilf), John Wiley & Sons, New York, 125–153.

Kaiser, H.F. (1963), Image analysis, *Problems in Measuring Change*, (edited by C. Harris), University of Wisconsin Press, Madison, Wis.

Kaiser, H.F., and J. Caffrey (1965), Alpha factor analysis, *Psychometrika*, **30**, 1–14.

Kendall, Maurice G., and Alan Stuart (1973), *The Advanced Theory of Statistics, Volume 2*: *Inference and Relationship*, 3rd ed., Charles Griffin & Company, London.

Kendall, Maurice G., and Alan Stuart (1979), *The Advanced Theory of Statistics*, Volume 2: *Inference and Relationship*, 4th ed., Oxford University Press, New York.

Kendall, Maurice G., Alan Stuart, and J. Keith Ord (1983), *The Advanced Theory of Statistics*, Volume 3: *Design and Analysis, and Time Series*, 4th. ed., Oxford University Press, New York.

Kennedy, William J., Jr. and James E. Gentle (1980), *Statistical Computing*, Marcel Dekker, New York.

Kinnucan, P., and H. Kuki (1968), *A Single Precision Inverse Error Function Subroutine*, Computation Center, University of Chicago.

Kirk, Roger E. (1982), *Experimental Design: Procedures for the Behavioral Sciences*, 2d ed., Brooks/Cole Publishing Company, Monterey, Calif.

Knuth, Donald E. (1981), *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms*, 2d ed., Addison-Wesley, Reading, Mass.

Lawley, D.N., and A.E. Maxwell (1971), *Factor Analysis as a Statistical Method*, 2d ed., Butterworth, London.

Learmonth, G.P., and P.A.W. Lewis (1973), *Naval Postgraduate School Random Number Generator Package LLRANDOM, NPS55LW73061A*, Naval Postgraduate School, Monterey, Calif.

Leavenworth, B. (1960), Algorithm 25: Real zeros of an arbitrary function, *Communications of the ACM*, **3**, 602.

Lehmann, E.L. (1975), *Nonparametrics: Statistical Methods Based on Ranks*, Holden-Day, San Francisco.

Levenberg, K. (1944), A method for the solution of certain problems in least squares, *Quarterly of Applied Mathematics*, **2**, 164–168.

Lewis, P.A.W., A.S. Goodman, and J.M. Miller (1969), A pseudorandom number generator for the System/360, *IBM Systems Journal*, **8**, 136–146.

Liepman, David S. (1964), Mathematical constants, *Handbook of Mathematical Functions*, Dover Publications, New York.

Lilliefors, H.W. (1967), On the Kolmogorov-Smirnov test for normality with mean and variance unknown, *Journal of the American Statistical Association*, **62**, 534–544.

Liu, J. W. H. (1986), On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, **12**, 249-264.

Liu, J. W. H. (1987), *A collection of routines for an implementation of the multifrontal method*, Technical Report CS-87-10, Department of Computer Science, York University, North York, Ontario, Canada.

Liu, J. W. H. (1989), The multifrontal method and paging in sparse Cholesky factorization. *ACM Transactions on Mathematical Software*, **15**, 310-325.

Liu, J. W. H. (1990), *The multifrontal method for sparse matrix solution: theory and practice*, Technical Report CS-90-04, Department of Computer Science, York University, North York, Ontario, Canada.

Longley, James W. (1967), An appraisal of least-squares programs for the electronic computer from the point of view of the user, *Journal of the American Statistical Association*, **62**, 819–841.

Maindonald, J.H. (1984), *Statistical Computation*, John Wiley & Sons, New York.

Mardia, K.V., J.T. Kent, J.M. Bibby (1979), *Multivariate Analysis*, Academic Press, New York.

Marquardt, D. (1963), An algorithm for least-squares estimation of nonlinear parameters, *SIAM Journal on Applied Mathematics*, **11**, 431–441.

Martin, R.S., and J.H. Wilkinson (1971), The Modified *LR* algorithm for complex Hessenberg matrices, *Volume II: Linear Algebra Handbook*, Springer, New York.

Micchelli, C.A. (1986), Interpolation of scattered data: Distance matrices and conditionally positive definite functions, *Constructive Approximation*, **2**, 11–22.

Micchelli, C.A., T.J. Rivlin, and S. Winograd (1976), The optimal recovery of smooth functions, *Numerische Mathematik*, **26**, 279–285.

Micchelli, C.A., Philip W. Smith, John Swetits, and Joseph D. Ward (1985), Constrained L_p approximation, *Constructive Approximation*, **1**, 93–102.

Müller, D.E. (1956), A method for solving algebraic equations using an automatic computer, *Mathematical Tables and Aids to Computation*, **10**, 208–215.

Milliken, George A., and Dallas E. Johnson (1984), *Analysis of Messy Data, Volume 1: Designed Experiments*, Van Nostrand Reinhold, New York.

Miller, Rupert G., Jr. (1980), *Simultaneous Statistical Inference*, 2d ed., Springer-Verlag, New York.

Moré, Jorge, Burton Garbow, and Kenneth Hillstrom (1980), *User Guide for MINPACK-1*, Argonne National Laboratory Report ANL 80–74, Argonne, Illinois.

Murtagh, Bruce A. (1981), *Advanced Linear Programming: Computation and Practice*, McGraw-Hill, New York.

Murty, Katta G. (1983), *Linear Programming*, John Wiley and Sons, New York.

Nelson, Peter (1989), Multiple Comparisons of Means Using Simultaneous Confidence Intervals, *Journal of Quality Technology*, **21**, 232–241.

Neter, John, and William Wasserman (1974), *Applied Linear Statistical Models*, Richard D. Irwin, Homewood, Ill.

Neter, John, William Wasserman, and Michael H. Kutner (1983), *Applied Linear Regression Models*, Richard D. Irwin, Homewood, Illinois.

Owen, D.B. (1962), *Handbook of Statistical Tables*, Addison-Wesley Publishing Company, Reading, Massachusetts.

Owen, D.B. (1965), A special case of the bivariate non-central *t* distribution, *Biometrika*, **52**, 437–446.

Parlett, B.N. (1980) *The Symmetric Eigenvalue Problem*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Petro, R. (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 624.

Piessens, R., E. deDoncker-Kapenga, C.W. Überhuber, and D.K. Kahaner (1983), *QUADPACK*, Springer-Verlag, New York.

Powell, M.J.D. (1978), A fast algorithm for nonlinearly constrained optimization calculations, in *Numerical Analysis Proceedings, Dundee 1977, Lecture Notes in Mathematics*, (edited by G. A. Watson), **630**, Springer-Verlag, Berlin, Germany, 144–157.

Powell, M.J.D. (1985), On the quadratic programming algorithm of Goldfarb and Idnani, *Mathematical Programming Study*, **25**, 46–61.

Powell, M.J.D. (1983), *ZQPCVX a FORTRAN subroutine for convex quadratic programming*, DAMTP Report 1983/NA17, University of Cambridge, Cambridge, England.

Reinsch, Christian H. (1967), Smoothing by spline functions, *Numerische Mathematik*, **10**, 177–183.

Rice, J.R. (1983), *Numerical Methods, Software, and Analysis*, Mcguire-Hill, New York.

Rietman, Edward (1989), *Exploring the Geometry of Nature*, Windcrest Books, Blue Ridge Summit, Pennsylvania.

Robinson, Enders A. (1967), *Multichannel Time Series Analysis with Digital Computer Programs*, Holden-Day, San Francisco.

Royston, J.P. (1982a), An extension of Shapiro and Wilk's *W* test for normality to large samples, *Applied Statistics*, **31**, 115–124.

Royston, J.P. (1982b), The *W* test for normality, *Applied Statistics*, **31**, 176–180.

Royston, J.P. (1982c), Expected normal order statistics (exact and approximate), *Applied Statistics*, **31**, 161–165.

Saad, Y., and M. H. Schultz (1986), GMRES: A generalized minimum residual algorithm for solving nonsymmetric linear systems, *SIAM Journal of Scientific and Statistical Computing*, **7**, 856-869.

Sallas, William M., and Abby M. Lionti (1988), *Some useful computing formulas for the nonfull rank linear model with linear equality restrictions*, IMSL Technical Report 8805, IMSL, Houston.

Savage, I. Richard (1956), Contributions to the theory of rank order statistics–the two-sample case, *Annals of Mathematical Statistics*, **27**, 590–615.

Schittkowski, K. (1980), Nonlinear programming codes, *Lecture Notes in Economics and Mathematical Systems*, **183**, Springer-Verlag, Berlin, Germany.

Schittkowski, K. (1983), On the convergence of a sequential quadratic programming method with an augmented Lagrangian line search function, *Mathematik Operations for Schung and Statistik, Serie Optimization*, **14**, 197–216.

Schittkowski, K. (1986), NLPQL: A FORTRAN subroutine solving constrained nonlinear programming problems, (edited by Clyde L. Monma), *Annals of Operations Research*, **5**, 485–500.

Schmeiser, Bruce (1983), Recent advances in generating observations from discrete random variates, *Computer Science and Statistics: Proceedings of the Fifteenth Symposium on the Interface*, (edited by James E. Gentle), North-Holland Publishing Company, Amsterdam, 154–160.

Schmeiser, Bruce W., and A.J.G. Babu (1980), Beta variate generation via exponential majorizing functions, *Operations Research*, **28**, 917–926.

Schmeiser, Bruce, and Voratas Kachitvichyanukul (1981), *Poisson Random Variate Generation*, Research Memorandum 81-4, School of Industrial Engineering, Purdue University, West Lafayette, Ind.

Schmeiser, Bruce W., and Ram Lal (1980), Squeeze methods for generating gamma variates, *Journal of the American Statistical Association*, **75**, 679–682.

Schwartz, G. (1978), Estimating the dimension of a model, *Ann. Statist*., 6, 461-464.

Searle, S.R. (1971), *Linear Models*, John Wiley & Sons, New York.

Shampine, L.F. (1975), Discrete least-squares polynomial fits, *Communications of the ACM*, **18**, 179–180.

Shampine, L.F., and C.W. Gear (1979), A user's view of solving stiff ordinary differential equations, *SIAM Review*, **21**, 1–17.

Singleton, R.C. (1969), Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **12**, 185–187.

Smith, B.T., J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, and C.B. Moler (1976), *Matrix Eigensystem Routines—EISPACK Guide*, Springer-Verlag, New York.

Smith, P.W. (1990), On knots and nodes for spline interpolation, *Algorithms for Approximation II*, J.C. Mason and M.G. Cox, Eds., Chapman and Hall, New York.

Snedecor, George W., and William G. Cochran (1967), *Statistical Methods*, 6th ed., Iowa State University Press, Ames, Iowa.

Spurrier, John D., and Steven P. Isham (1985), Exact simultaneous confidence intervals for pairwise comparisons of three normal means, *Journal of the American Statistical Association*, **80**, 438–442.

Stewart, G.W. (1973), *Introduction to Matrix Computations*, Academic Press, New York.

Stoer, J. (1985), Principles of sequential quadratic programming methods for solving nonlinear programs, *Computational Mathematical Programming*, (edited by K. Schittkowski), NATO ASI Series, **15**, Springer-Verlag, Berlin, Germany.

Stoline, Michael R. (1981), The status of multiple comparisons: simultaneous estimation of all pairwise comparisons in one-way ANOVA designs, *The American Statistician*, **35**, 134–141.

Strecok, Anthony J. (1968), On the calculation of the inverse of the error function, *Mathematics of Computation*, **22**, 144–158.

Stroud, A.H., and D.H. Secrest (1963), *Gaussian Quadrature Formulae*, Prentice-Hall, Englewood Cliffs, New Jersey.

Temme, N.M. (1975), On the numerical evaluation of the modified Bessel function of the third kind, *Journal of Computational Physics*, **19**, 324–337.

Thompson, I.J., and A.R. Barnett (1987), Modified Bessel functions $I_v(z)$ and $K_v(z)$ of real order and complex argument, to selected accuracy, *Computer Physics Communication*, **47**, 245–257.

Tukey, John W. (1962), The future of data analysis, *Annals of Mathematical Statistics*, **33**, 1–67.

Velleman, Paul F., and David C. Hoaglin (1981), *Applications, Basics, and Computing of Exploratory Data Analysis*, Duxbury Press, Boston.

Walker, H. F. (1988), Implementation of the GMRES method using Householder transformations, *SIAM Journal of Scientific and Statistical Computing*, **9**, 152-163.

Watkins, David S., L. Elsner (1991), Convergence of algorithm of decomposition type for the eigenvalue problem, *Linear Algebra Applications*, **143**, 19–47.

Weisberg, S. (1985), *Applied Linear Regression*, 2nd edition, John Wiley & Sons, New York.

B

APPENDIX

Summary of Routines

date and the maturity date.

COUPON_PCD Function page 467

Evaluates the coupon date which immediately precedes the settlement date.

CSINTERP Function page 111

Computes a cubic spline interpolant, specifying various endpoint conditions. The default interpolant satisfies the not-a-knot condition.

CSSHAPE Function page 116

Computes a shape-preserving cubic spline.

CSSMOOTH Function page 159

Computes a smooth cubic spline approximation to noisy data by using cross-validation to estimate the smoothing parameter or by directly choosing the smoothing parameter.

CUM_INTR Function page 399

Evaluates the cumulative interest paid between two periods.

CUM_PRINC Function page 401

Evaluates the cumulative principal paid between two periods.

DATETODAYS Function page 559

Computes the number of days from January 1, 1900, to the given date.

DAYSTODATE Procedure page 558

Gives the date corresponding to the number of days since January 1, 1900.

DEPREC_AMORDEGRC Function page 449

Evaluates the depreciation for each accounting period. During the evaluation of the function a depreciation coefficient based on the asset life is applied.

DEPREC_AMORLINC Function page 450

Evaluates the depreciation for each accounting period.

DEPRECIATION_DB Function page 402

Evaluates the depreciation of an asset using the fixed-declining balance method.

DEPRECIATION_DDB Function page 404

Evaluates the depreciation of an asset using the double-declining balance method.

DEPRECIATION_SLN Function page 406

ELE Function page 382

Evaluates the complete elliptic integral of the second kind E(x).

ELK Function page 381

Evaluates the complete elliptic integral of the kind K(x).

ELRC Function page 387

Evaluates an elementary integral from which inverse circular functions, logarithms and inverse hyperbolic functions can be computed.

ELRD Function page 384

Evaluates Carlson's elliptic integral of the second kind $R_D(x, y, z)$.

ELRF Function page 383

Evaluates Carlson's elliptic integral of the first kind $R_F(x, y, z)$.

ELRJ Function page 386

Evaluates Carlson's elliptic integral of the third kind $R_J(x, y, z, \rho)$

ERF Function page 356

Evaluates the real error function erf(x). Using a keyword, the inverse error function $erf^{-1}(x)$ can be evaluated.

ERFC Function page 358

Evaluates the real complementary error function erf(x). Using a keyword, the inverse complementary error function $erf^{-1}(x)$ can be evaluated.

FAURE_INIT Function page 524

Initializes the structure used for computing a shuffled Faure sequence.

FAURE_NEXT_PT Function page 528

Generates a shuffled Faure sequence.

FCDF Function page 542

Evaluates the F distribution function. Using a keyword, the inverse of the F distribution function can be evaluated.

FCN_DERIV Function page 224

Computes the first, second, or third derivative of a user-supplied function.

FCNLSQ Function page 141

Computes a least-squares fit using user-supplied functions.

FFTCOMP Function page 273

Computes the discrete Fourier transform of a real or complex sequence. Using keywords, a real-to-complex transform or a two-dimensional complex Fourier transform can be computed.

period.

Solves a linear least-squares problem with linear constraints.

NONLINPROG Function page 338

Solves a general nonlinear programming problem using the successive quadratic programming (QP) algorithm.

NORM Function page 570

Computes various norms of a vector or the difference of two vectors.

NORMALCDF Function page 534

Evaluates the standard normal (Gaussian) distribution function. Using a keyword, the inverse of the standard normal (Gaussian) distribution can be evaluated.

NUM_PERIODS Function page 426

Evaluates the number of periods for an investment for which periodic and constant payments are made and the interest rate is constant.

ODE Function page 534

Solves an initial value problem, which is possibly stiff, using the Adams-Gear methods for ordinary differential equations. Using keywords, the Runge-Kutta-Verner fifth-order and sixth-order method can be used if the problem is known not to be stiff.

PAYMENT Function page 427

Evaluates the periodic payment for an investment.

PDE_MOL Function page 245

Solves a system of partial differential equations of the form $ut = f(x, t, u, u_x, u_{xx})$ using the method of lines. The solution is represented with cubic Hermite polynomials.

POISSON2D Function page 263

Solves Poisson's or Helmholtz's equation on a two-dimensional rectangle using a fast Poisson solver based on the HODIE finite-difference scheme on a uniform mesh.

POISSONCDF Function page 555

Evaluates the Poisson distribution function.

PRES_VAL_SCHD Function page 430

Evaluates the present value for a schedule of cash flows. It is not necessary that the cash flows be periodic.

PRESENT_VALUE Function page 429

Evaluates the net present value of a stream of equal periodic cash flows, which are subject to a given discount rate..

PRICE_MATURITY Function page 472

SP_BDPDFAC Function page 74

Compute the R^TR Cholesky factorization of symmetric positive definite matrix, A, in band symmetric storage mode.

SP_BDPDSOL Function page 72

Solve a symmetric positive definite system of linear equations $Ax = b$ in band symmetric storage mode.

SP_BDSOL Function page 59

Solve a general band system of linear equations $Ax = b$.

SP_CG Function page 79

Solve a real symmetric definite linear system using a conjugate gradient method. Using keywords, a preconditioner can be supplied.

SP_GMRES Function page 76

Solve a linear system $Ax = b$ using the restarted generalized minimum residual (GMRES) method.

SP_LUFAC Function page 54

Compute an LU factorization of a sparse matrix stored in either coordinate format or CSC format.

SP_LUSOL Function page 49

Solve a sparse system of linear equations $Ax = b$.

SP_MVMUL Function page 82

Compute a matrix-vector product involving sparse matrix and a dense vector.

SP_PDFAC Function page 68

Solve a sparse symmetric positive definite system of linear equations $Ax = b$.

SP_PDSOL Function page 65

Solve a sparse symmetric positive definite system of linear equations $Ax = b$.

SPINTEG Function page 137

Computes the integral of a one- or two-dimensional spline.

SPVALUE Function page 132

Computes values of a spline or values of one of its derivatives.

SVDCOMP Function page 36

Computes the singular value decomposition (SVD), $A=USV^T$, of a real or complex rectangular matrix A. An estimate of the rank of A also can be computed.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z